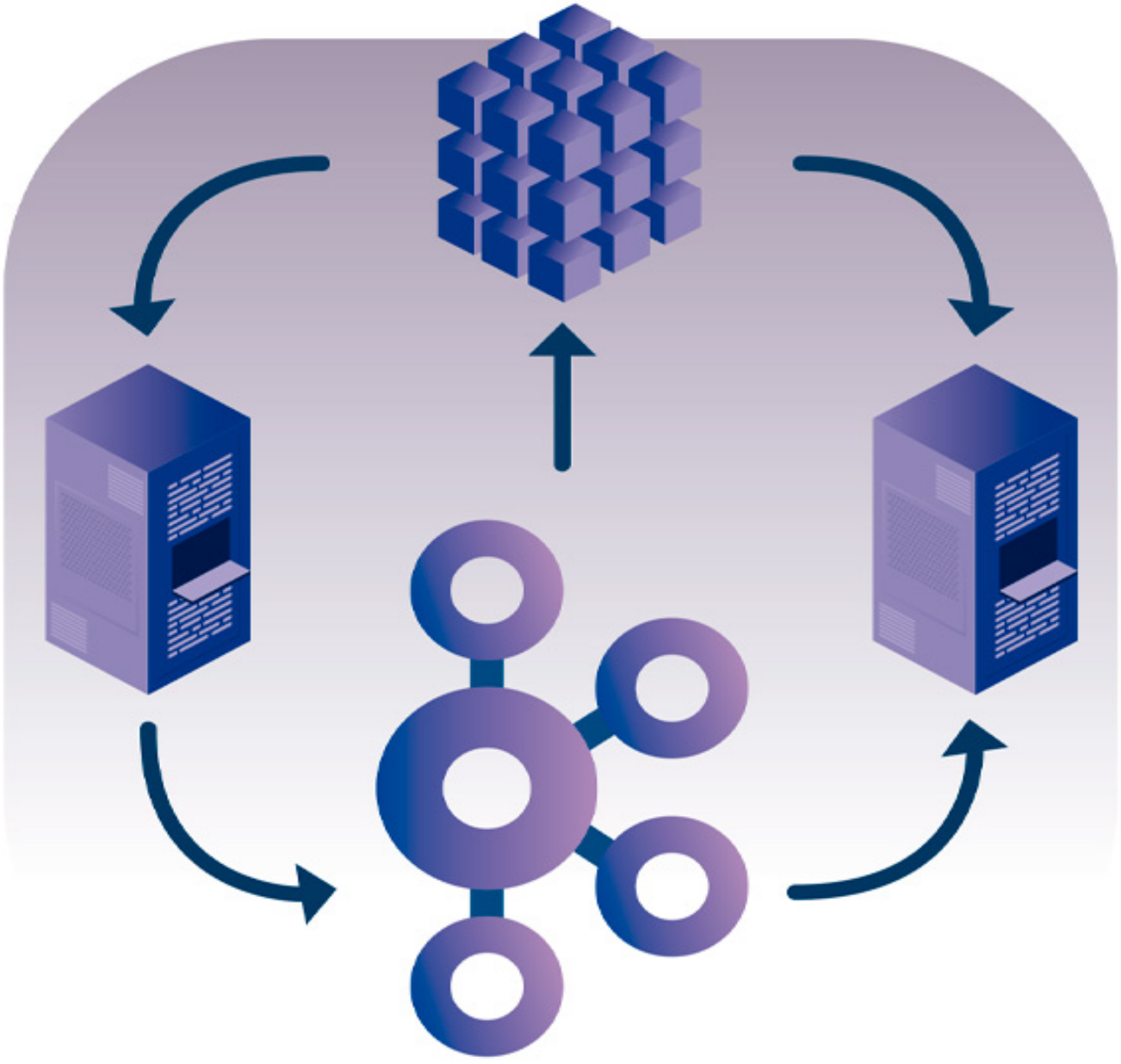
Kafka e Schema Registry

Modelagem de eventos com robustez e segurança



Casa do Código | alura

RODRIGO VIEIRA PINTO

Sumário

- ISBN
- Agradecimentos
- Sobre o livro
- Sobre o autor
- 1. O Alura Eats - um novo delivery de comida
- 2. Sem controle, sem garantia: a falta que o Schema Registry faz
- 3. Explorando a API do Schema Registry: Endpoints, Subjects e Schemas
- 4. Implementando Schema Registry no consumidor de mensagens
- 5. O Avro como formato de mensagens
- 6. Consumidores de mensagens Avro e o AVDL
- 7. Avançando com Schema Registry: Modos de compatibilidade, tipos lógicos do Avro e customizações
- 8. O Protobuf como formato de mensagens
- 9. Integração com o Kafka Connect
- 10. Conclusão
- 11. Bibliografia

ISBN

Impresso: 978-85-5519-402-3

Digital: 978-85-5519-401-6

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Pinto, Rodrigo Vieira
Kafka e Schema registry : modelagem de eventos com robustez e segurança / Rodrigo Vieira Pinto. -- 1. ed. -- São Paulo : AOVS Sistemas de Informática, 2025.

ISBN 978-85-5519-402-3

1. Aplicativos - Software - Desenvolvimento 2. Ciência da computação 3. Linguagem de programação para computadores 4. Web sites - Desenvolvimento I. Título.

25-284624 CDD-005.1068

Índices para catálogo sistemático:

1. Software : Desenvolvimento : Projetos : Gerenciamento : Ciência da computação 005.1068

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Primeiramente, agradeço a Deus por me proporcionar saúde, sabedoria, perseverança e oportunidades para crescer cada vez mais — e, com isso, poder transmitir o conhecimento adquirido de todas as formas possíveis.

Um agradecimento especial também a todo o time de arquitetos e desenvolvedores da VR, pelas diversas discussões que tivemos sobre como escrever sistemas melhores e arquiteturas orientadas a eventos. Sem essas trocas, este livro não teria sido concebido.

E, como não poderia deixar de ser, minha gratidão mais profunda à Marcia e ao Leonardo: conseguimos de novo!

Obrigado a você também, que adquiriu este trabalho — espero, sinceramente, que ele ajude você a escrever sistemas cada vez melhores.

Sobre o livro

O Kafka se destaca como uma das ferramentas mais impactantes e versáteis na computação distribuída. Sua capacidade de lidar com grandes volumes de dados, em tempo real e com alta resiliência, transformou a maneira como sistemas modernos interagem, trocam informações e escalam suas operações. Desde pipelines de dados robustos até sistemas de eventos que conectam microsserviços, o Kafka se tornou um pilar essencial em arquiteturas contemporâneas.

No entanto, como em qualquer tecnologia poderosa, trabalhar com Kafka apresenta seus próprios desafios. Um dos mais comuns — e críticos — é a necessidade de garantir que os dados enviados e consumidos pelos tópicos mantenham consistência e estrutura ao longo do tempo. Imagine, por exemplo, uma aplicação que consome dados de um tópico e, de repente, recebe uma mensagem com um formato inesperado. Esse pequeno desvio pode causar erros imprevisíveis, comprometer integrações e até mesmo gerar indisponibilidade em sistemas *downstream*.

Um sistema *downstream* recebe dados ou depende de outro sistema para funcionar. Em uma arquitetura que possui o Kafka como um dos seus componentes, temos o sistema produtor de mensagens (que pode ser chamado de *upstream*) e o que consome as mensagens, chamado de *downstream*.

Ao confiar exclusivamente no Kafka para o transporte de mensagens, muitas equipes enfrentam dificuldades em gerenciar a evolução dos formatos de dados (*schemas*), assegurando compatibilidade entre versões e evitando quebras nos sistemas que dependem dessas informações. Esse cenário não apenas complica o desenvolvimento, como também adiciona complexidade à operação e manutenção de *pipelines* de dados.

É aqui que o Schema Registry entra em cena. Essa ferramenta permite que desenvolvedores, operadores e arquitetos registrem, validem e versionem os *schemas* de mensagens que trafegam nos tópicos Kafka. Com essa solução, as mensagens não são mais um "mistério" que depende de contratos informais ou comunicação verbal entre equipes. Em vez disso, o *schema* se torna uma fonte de verdade centralizada e confiável, trazendo ordem e

previsibilidade para ambientes onde mudanças são inevitáveis.

Este livro foi pensado para ajudar você a compreender e aproveitar melhor o uso do Schema Registry e transformar a forma como trabalha com o Kafka. Ao longo das próximas páginas, você aprenderá a integrar essa poderosa ferramenta ao seu fluxo de trabalho, entenderá como evoluir *schemas* de forma segura e garantirá que a comunicação entre os serviços seja sempre consistente.

Seja você engenheiro(a) de dados, arquiteto(a) de sistemas ou desenvolvedor(a) *back-end*, este livro oferece o conhecimento necessário para resolver um dos maiores desafios ao trabalhar com sistemas distribuídos baseados em Kafka.

Bem-vindo(a) a esta jornada. Ao longo das páginas seguintes, você aprenderá como o Schema Registry fortalece as integrações com Kafka, trazendo mais previsibilidade e segurança para seus *pipelines* de dados. Com esse conhecimento, você estará preparado(a) para construir soluções escaláveis, resilientes e prontas para o futuro.

Sobre o autor

Rodrigo Vieira Pinto é mestre em Engenharia de Software pelo IPT-SP (Instituto de Pesquisas Tecnológicas de São Paulo) e pós-graduado em Engenharia de Software pela PUC-SP (Pontifícia Universidade Católica - SP). Possui cerca de 26 anos de experiência em desenvolvimento de sistemas em Java, PHP e Ruby, passando também pela linguagem C. Já trabalhou com servidores de aplicação como JBoss e Websphere, usando *frameworks* como EJB 2 e 3, JSP/Servlet, JSF e Hibernate (e ficou feliz quando o Spring Boot surgiu). Gosta muito de paradigmas de programação, padrões de projeto, testes automatizados, de código simples que funciona e de compartilhar conhecimento.

Ministra aulas de programação desde 2018, tanto em cursos livres quanto em faculdades, passando por Caelum, Grupo Alura, SPTech (São Paulo Tech School) e FIAP (Faculdade de Informática e Administração Paulista). Atualmente é Chapter Lead de Arquitetura e Backend da VR Benefícios e professor da disciplina de Domain-Driven Design with Java na FIAP.

Nas horas vagas, estuda tecnologias alternativas, história do Brasil e religião. Também pratica esportes

como natação e corrida, e gosta de assistir a corridas de automóvel.

Amante de *software* bem escrito, entende que a tecnologia da informação tem muito a crescer e evoluir — e que nós, muito a contribuir.

Capítulo 1

O Alura Eats - um novo delivery de comida

Alura Eats é uma *startup* fictícia criada com o objetivo de melhorar os serviços de *delivery* de comida. Nós, desenvolvedores desta empresa, notamos que as companhias do mesmo segmento não conseguem atuar em todos os pontos do país, devido às suas tarifas cada vez mais caras. Buscamos, portanto, oferecer preços competitivos para os restaurantes.

Por ser uma empresa relativamente nova, usaremos o que há de mais moderno nas nossas aplicações — tanto em ferramentas quanto em decisões arquiteturais.

O Alura Eats terá basicamente três tipos de usuários: os responsáveis pelos estabelecimentos, os entregadores e os clientes. O responsável pelo restaurante usará uma solução chamada de *ms-restaurantes* e, com ela, poderá:

- cadastrar os cardápios;
- adicionar ou atualizar os itens de cardápio.

Os clientes poderão:

- visualizar cardápios;
- selecionar itens de cardápio;
- realizar pedidos.

Para isso, utilizarão o *ms-pedidos*.

Já os entregadores utilizarão os serviços do *ms-entregadores* para:

- receber avisos de novos pedidos;
- realizar a entrega desses pedidos.

Por último, tanto os restaurantes quanto os clientes receberão notificações sobre o *status* dos pedidos por meio do *ms-notificacoes*.

Ao longo deste livro, descreveremos apenas o *back-end* da solução. Assim, todas as entradas das aplicações serão construídas com interfaces REST ou mesmo consumidores ou produtores de mensagens Kafka. Tomaremos contato com a solução incrementalmente, utilizando não somente código, mas também diagramas que nos ajudarão a ter uma visão melhor do todo.

1.1 Ferramentas a serem instaladas

Caso você queira acompanhar os exemplos de código deste livro no seu computador, é fundamental que as seguintes ferramentas sejam instaladas:

- Git;
- JDK (Java Development Kit) versão 21 ou superior;
- Maven (versão 3.9.6 ou mais recente);
- Uma IDE (*Integrated Development Environment*) Java de sua preferência, com suporte ao Maven (como Eclipse, IntelliJ etc.);
- um *client* de banco de dados para MongoDB de sua preferência. No livro, usaremos o Studio 3T, mas fique à vontade para usar o *client* de sua escolha;
- Docker (para rodarmos o Kafka, o Schema Registry, o MongoDB e outras ferramentas). Usaremos o Kafka e o Schema Registry disponibilizados pela Confluent;
- Um *REST client*, como o Insomnia ou o Postman (este último será usado nos exemplos deste livro). Há também um projeto que pode ser importado para o Postman, o `livro_schema_registry.postman_collection.json`. Dessa maneira, será possível testar as chamadas REST dos microsserviços necessários da solução.

Usaremos o Redpanda Console como *client* do Kafka e do Schema Registry. Posto isso, poderemos visualizar os tópicos, suas mensagens e os *schemas* de cada um deles. Assim como o Kafka, ele será instalado via `docker-compose.yml`. E, assim como o Postman, fique à vontade para escolher outro *client*. Entretanto, se sua escolha for outra, verifique se ele também possui suporte à integração com o Schema Registry.

1.2 Códigos usados por todo o livro

Os códigos apresentados por todo o livro estão em https://github.com/rodrigovp/livro-schema-registry. A solução descrita será evoluída ao longo dos capítulos, e para cada capítulo há uma *branch* (exceto este, que está na *main*). Os códigos do capítulo 2 estão na *branch* `cap-2`, do capítulo 3 em `cap-3` e assim por diante. No início de cada capítulo, as *branches* serão referenciadas novamente para facilitar a busca dos códigos correspondentes.

Vamos começar?

Capítulo 2

Sem controle, sem garantia: a falta que o Schema Registry faz

> Os códigos deste capítulo encontram-se na *branch* `cap-2`.

Assim que os restaurantes se cadastram no Alura Eats, já podem criar seu cardápio no sistema. Para isso, acessam o *ms-restaurantes* e inserem as informações de cada item. Se o item for o primeiro cadastrado, um novo cardápio é criado para o restaurante, e o item é então associado a esse menu.

Essa operação é realizada por meio do acesso a um *endpoint* REST, utilizando o comando cURL a seguir. A mensagem contém os seguintes campos:

- `id-restaurante` - um id que identifica tanto o restaurante quanto o cardápio;
- `numero` - o número do item de cardápio;
- `nome` - o nome do prato;
- `descricao` - a descrição do prato.

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
```

```
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa",
    "descricao": "Uma feijoada onde o céu é o limite"
}'
```

Uma vez que o cardápio é criado ou atualizado, uma mensagem é enviada ao tópico Kafka `restaurantes.cardapio.itens`, contendo o item recém-cadastrado. Cada mensagem possui uma chave com dois campos, conforme o formato a seguir:

```
{
   "idRestaurante":1,
   "idItem":1
}
```

Os campos da mensagem contêm apenas os campos correspondentes ao item de cardápio:

```
{
   "nome":"Feijoada Completa",
   "descricao":"Uma feijoada onde o céu é o limite"
}
```

Há também um microsserviço chamado *ms-pedidos*, que consome as mensagens desse tópico e as persiste em uma base de dados MongoDB. Essa base contém os menus dos restaurantes, ou seja, trata-se de uma base para os cardápios.

Também são enviados no `header` os nomes das classes que geram tanto a chave da mensagem quanto o seu valor, e é possível vê-los na mensagem, como na figura:

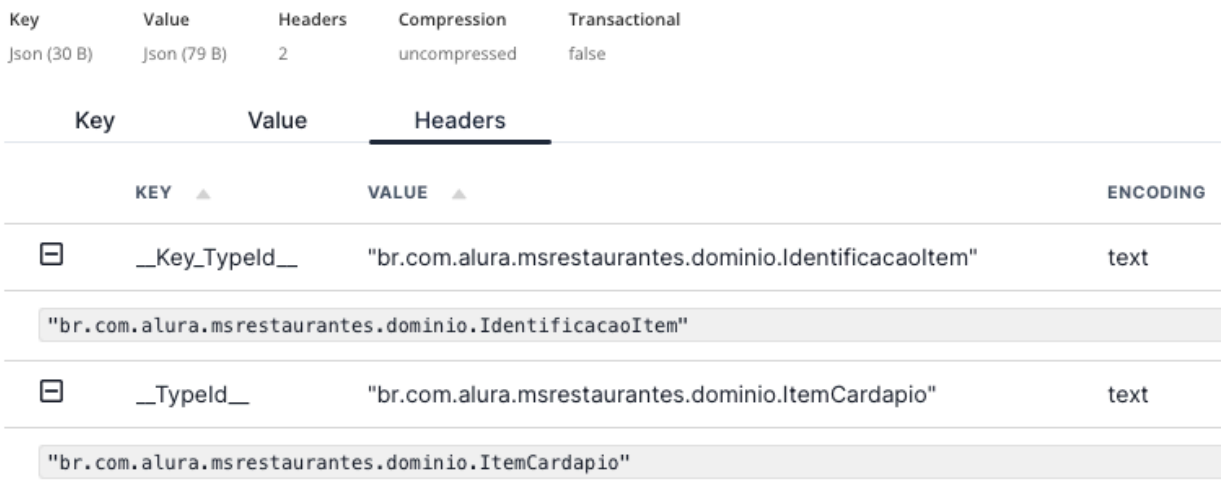


Figura 2.1: Headers enviados em cada mensagem.

Por padrão, o `header` é lido por consumidores que trabalham com Spring Kafka para facilitar o *binding* das mensagens. Entretanto, não usaremos essa informação no *ms-pedidos*, conforme veremos posteriormente. A solução até o momento é representada na próxima figura:

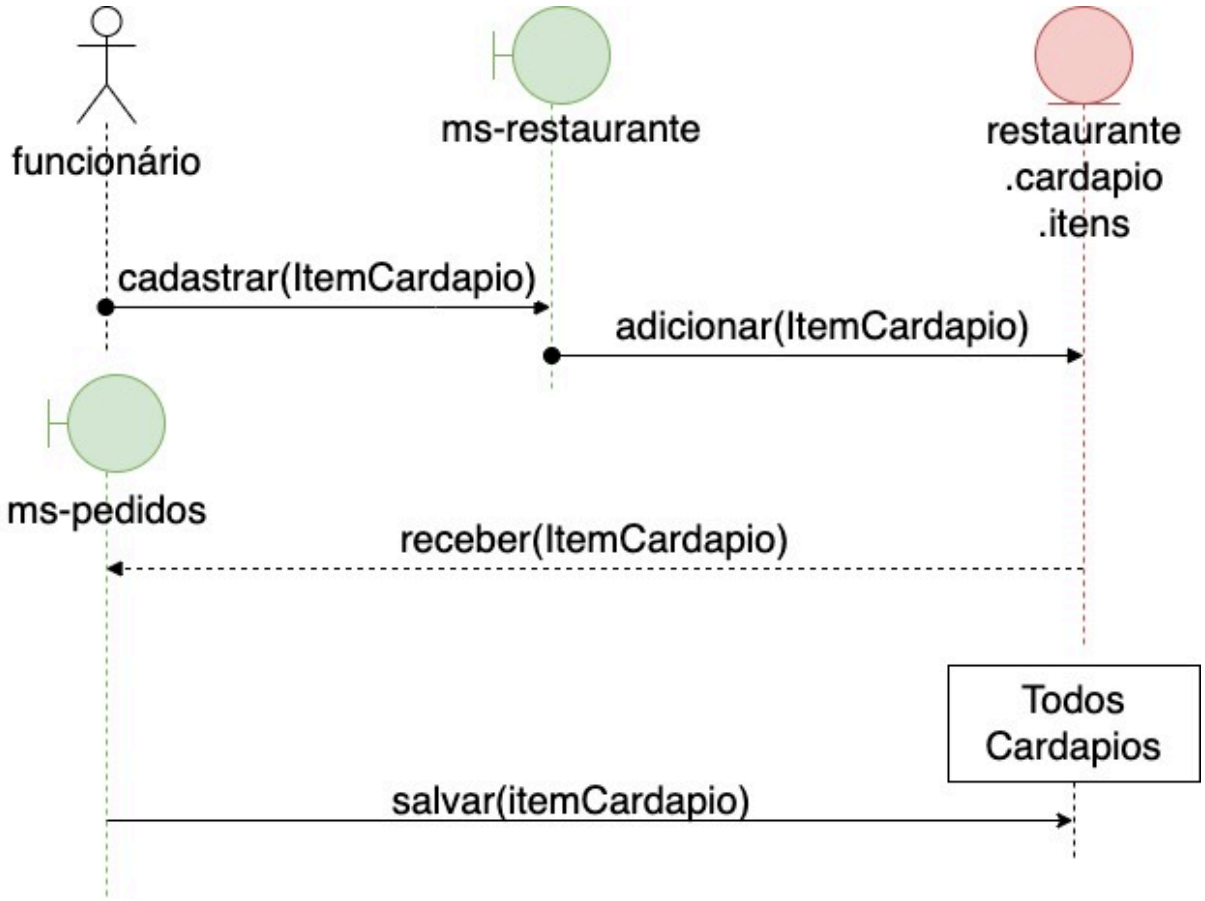


Figura 2.2: Cadastro de um item de cardápio no Alura Eats.

Além disso, no *ms-pedidos*, o *binding* das mensagens é feito para classes existentes dentro do próprio microsserviço. Elas são semelhantes às classes do *ms-restaurantes*, mas em pacotes diferentes. Dessa forma, é possível que as classes de cada um dos microsserviços evoluam de forma independente.

Para que isso seja possível, os desserializadores de chave e valor usam instâncias de `JsonDeserializer`, configuradas para ignorar os *headers* da mensagem. Esses *headers* informam as classes que deveriam ser

usadas no *binding*. Para isso, o segundo argumento no construtor é o `boolean` *false*. Se fosse *true*, os *headers* seriam levados em consideração. Veja a instanciação do `JsonDeserializer`:

```
private <T> JsonDeserializer<T>
jsonDeserializer(Class<T> classes){
    return new JsonDeserializer<T>(classes, false);
}
```

> O primeiro argumento do construtor declarado é a classe que se deseja usar no *binding*.

O método mencionado pode ser utilizado tanto para desserializar a chave quanto o valor da mensagem.

O código completo da classe `KafkaObjects` no *ms-pedidos* está na sequência:

```
package br.com.alura.mspedidos.infra.kafka;

import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import
org.apache.kafka.clients.consumer.ConsumerConfig;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.kafka.config.ConcurrentKafkaListene
```

```
rContainerFactory;
import
org.springframework.kafka.config.KafkaListenerContainer
Factory;
import
org.springframework.kafka.core.DefaultKafkaConsumerFact
ory;
import
org.springframework.kafka.listener.ConcurrentMessageLis
tenerContainer;
import
org.springframework.kafka.support.serializer.JsonDeseri
alizer;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaObjects {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.consumer.group-id}")
    private String groupId;

    @Value("${spring.kafka.consumer.auto-offset-
reset}")
    private String autoOffsetReset;

    @Bean

KafkaListenerContainerFactory<ConcurrentMessageListener
Container<IdentificacaoItem, ItemCardapioRecebido>>
kafkaListener() {
```

```
ConcurrentKafkaListenerContainerFactory<IdentificacaoIt
em, ItemCardapioRecebido> factory =
                new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(
                consumerConfigs(),

jsonDeserializer(IdentificacaoItem.class),

jsonDeserializer(ItemCardapioRecebido.class)
        ));
        return factory;
    }

    private <T> JsonDeserializer<T>
jsonDeserializer(Class<T> classe){
        return new JsonDeserializer<T>(classe, false);
    }

    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
groupId);

props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset);

props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI
G, JsonDeserializer.class);
```

```
        return props;
    }
}
```

2.1 Mensagens sem contrato definido e seus problemas

Como vimos na figura anterior, usaremos o Kafka como sistema de mensageria entre o *ms-restaurantes* e o *ms-pedidos*, sendo o primeiro o produtor de mensagens, e o segundo o consumidor.

As mensagens podem ser de diferentes formatos. O Kafka não restringe os formatos possíveis. Elas podem, inclusive, não possuir um formato definido ou estruturado. No entanto, é possível afirmar que os formatos mais usados atualmente são JSON, AVRO e Protobuf.

Enquanto JSON é um formato baseado em texto, tanto AVRO quanto Protobuf são protocolos binários. Protocolos binários são, dentre outras características, mais facilmente processáveis pelos computadores e geram mensagens de tamanho final menor se comparados aos baseados em texto. Já os protocolos baseados em texto são mais fáceis de serem entendidos por seres humanos. Os exemplos deste livro usam apenas mensagens estruturadas nos exemplos de código.

As mensagens podem atender inicialmente aos propósitos de comunicação de produtores e consumidores. Mas, com o passar do tempo e as constantes modificações pelas quais um *software* precisa passar, essas aplicações podem enfrentar alguns problemas, especialmente no que diz respeito à comunicação entre elas:

- o produtor pode adicionar um novo campo opcional, mas o consumidor não foi atualizado para entender esse novo campo, que pode ser crítico para uma nova regra de negócio ou auditoria. Como resultado, o consumidor pode falhar ao processar a mensagem ou ignorar dados importantes. E se esse campo for adicionado como obrigatório?

- o produtor remove ou renomeia um campo existente que o consumidor espera. O consumidor pode lançar exceções ao tentar acessar um campo que não existe mais ou ao processar o dado de maneira errada, pois o nome ou tipo do campo mudou.
- o produtor altera o tipo de um campo, sem que os consumidores sejam atualizados. É bem provável que os consumidores deixem de processar mensagens até que a atualização seja feita neles também.

É possível fazermos uma gestão manual de todos os consumidores. Nesse caso, os responsáveis pela aplicação produtora precisam notificar todos os responsáveis pelas aplicações consumidoras de mudanças no formato da mensagem. Assim, todos estarão usando o mesmo esquema. Em pequenas empresas, com poucas aplicações, isso pode não ser tão difícil. Mas quando se possui muitas aplicações, essa tarefa pode ser bem desafiadora. Uma pequena inconsistência na versão do esquema usado pode resultar em falhas de integração.

2.2 O Schema Registry

Para que esses problemas não ocorram, é necessário que os esquemas das mensagens estejam sob um controle centralizado e padronizado, que permita a sua evolução e que seja acessível tanto por produtores quanto por consumidores.

Quando falamos de integrações REST, este papel é atribuído à aplicação que disponibiliza a API a ser acessada. Já quando falamos de dados em banco de dados, podemos pensar em *constraints*, tipos associados às colunas e chaves primária e estrangeira. Mas quando falamos de Kafka, este papel é atribuído ao Schema Registry.

O Schema Registry é, portanto, uma ferramenta que permite o gerenciamento de esquemas de mensagens que irão para os tópicos. Tanto os produtores quanto os consumidores devem consultá-lo antes de preparar ou de receber as mensagens. Assim, eles saberão exatamente o seu formato, e poderão tanto escrever quanto ler todos os campos corretamente.

Problemas no preenchimento dos itens de cardápio

Com as aplicações da maneira com que estão construídas atualmente, é perfeitamente possível deixarmos de preencher alguns campos na chamada

ao *ms-restaurantes*. Considere o exemplo a seguir. Note que o corpo da mensagem não contém o campo `descricao`:

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa"
}'
```

Uma mensagem dessas enviada ao *ms-restaurantes* produz o seguinte resultado no tópico:

```
{
   "nome":"Feijoada Completa",
   "descricao":NULL
}
```

Isso significa que agora temos um item de cardápio com informações faltantes! Poderíamos inserir validações no *controller* do *ms-restaurantes* que impedissem o recebimento de requisições com problemas. Entretanto, com essa solução, caso outros produtores passassem a produzir mensagens para o mesmo tópico, eles também teriam que ter esse cuidado.

Poderíamos também inserir validadores nos consumidores de mensagens. Mas isso implica que todos os consumidores devem possuir validadores de

mensagem, resultando em possível código repetido em diversas aplicações.

Poderíamos inclusive deixar de enviar o `id-restaurante` na mensagem, fazendo a seguinte chamada ao *ms-restaurantes*:

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
--data '{
    "numero": 1,
    "nome": "Feijoada Completa",
    "descricao": "Uma feijoada onde o céu é o limite"
}'
```

Nesse caso, o problema é ainda mais grave. Pelo fato de estarmos trabalhando com uma variável de tipo `int` para representar o `id-restaurante`, caso nenhum valor tenha sido atribuído a ela, será atribuído o valor 0 (zero). Isso significa que é criado um cardápio com `id` igual a zero!

Vejamos no banco de dados:

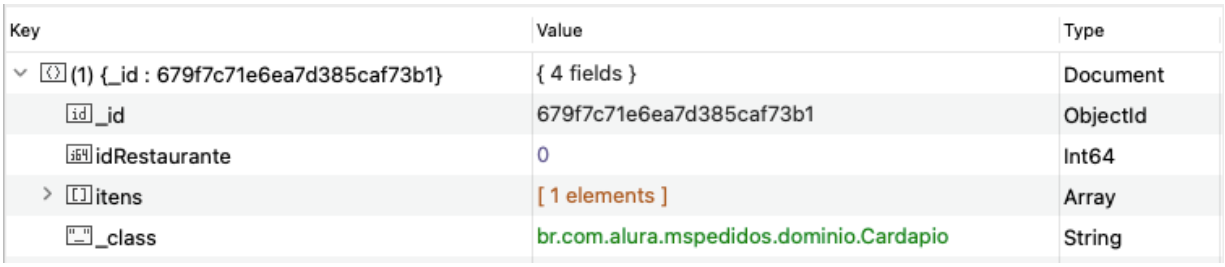


Figura 2.3: Cardápio com identificação inválida.

Na linguagem Java, caso algum tipo primitivo tenha sido declarado e não inicializado, é assumido um valor *default* para ele. Logo:

- para tipos numéricos como `int`, `byte` ou `short`, o valor *default* é 0 (zero);
- para tipos como `double` ou `float`, os valores são 0.0d e 0.0f respectivamente.

Para mais informações, consulte https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

2.3 Inserindo o Schema Registry na nossa solução

Para resolvermos esses problemas, declaramos no arquivo `docker-compose.yml` a subida de um Schema Registry. O Schema Registry possui diversas variáveis de ambiente, algumas obrigatórias, outras não. Nesse momento, vamos utilizar apenas um conjunto mínimo necessário para os nossos propósitos:

- `SCHEMA_REGISTRY_HOST_NAME`: um nome de rede para a aplicação;

- `SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS`: URLs do *cluster* Kafka que serão observados pelo Schema Registry;

A configuração completa do Schema Registry no `docker-compose.yml` está a seguir:

```
    schema-registry:
      image: confluentinc/cp-schema-registry:7.9.1
      hostname: schema-registry
      container_name: schema-registry
      depends_on:
          - kafka
      ports:
          - "8081:8081"
      environment:
          SCHEMA_REGISTRY_HOST_NAME: schema-registry
          SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS:
kafka:29092
      networks:
          bubble:
```

Também será necessário incluir o suporte ao Schema Registry no Redpanda Console. Para isso, precisamos declarar:

- a URL do Schema Registry;
- se ele está habilitado para ser acessado.

Essas configurações devem ser escritas como a seguir, logo depois da declaração da URL do Kafka:

```
schemaRegistry:
  enabled: true
  urls: ["http://schema-registry:8081"]
```

Por último, vamos inserir a dependência do Redpanda Console do Schema Registry, assim como fazemos com o Kafka:

```
depends_on:
  - kafka
  - schema-registry
```

A configuração completa do Redpanda Console, já com a inclusão do suporte ao Schema Registry, está a seguir:

```
redpanda:
  image:
docker.redpanda.com/redpandadata/console:v2.5.2
  container_name: redpanda
  ports:
    - 18080:8080
  depends_on:
      - kafka
      - schema-registry
  restart: always
  entrypoint: /bin/sh
  command: -c "echo \"$$CONSOLE_CONFIG_FILE\" >
/tmp/config.yml; /app/console"
  environment:
      CONFIG_FILEPATH: /tmp/config.yml
      CONSOLE_CONFIG_FILE: |
        kafka:
          brokers: ["kafka:29092"]
```

```
          schemaRegistry:
            enabled: true
            urls: ["http://schema-registry:8081"]
  networks:
    bubble:
```

2.4 Inserindo um schema para a mensagem

Uma possível maneira de cadastrar um *schema* para o tópico que desejamos controlar é por meio do envio de uma mensagem para esse tópico. A mensagem será enviada para o tópico e um *schema* será cadastrado sem problemas.

Para que isso seja possível, precisamos adicionar no `pom.xml` do *ms-restaurantes* a biblioteca `kafka-json-schema-serializer`. Ela é responsável por fornecer classes que permitem ao nosso microsserviço se comunicar com o Schema Registry, podendo acessar o *schema* do tópico que desejamos trabalhar, verificar versões de schemas, entre outras funcionalidades.

Segue o XML que deve ser adicionado ao `pom`:

```
<properties>
    ...
    <confluent.version>7.9.1</confluent.version>
</properties>
```

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-json-schema-
serializer</artifactId>
    <version>${confluent.version}</version>
</dependency>
```

> Um *schema* pode ter versões como em uma API Rest. Discutiremos esse assunto em detalhes no capítulo 7.

Para que essas e outras dependências da Confluent sejam baixadas, precisamos também inserir o repositório da empresa:

```
<repositories>
    <repository>
        <id>confluent</id>
        <name>Confluent</name>
        <url>https://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

> O repositório será inserido em todos os microsserviços daqui em diante, visto que todos precisarão de dependências provenientes da Confluent.

Devemos também substituir o `JsonSerializer` como serializador das mensagens pela classe

`KafkaJsonSchemaSerializer`. Essa classe pertence à biblioteca já mencionada e realizará, de fato, a conexão entre o *ms-restaurantes* e o Schema Registry.

Com a substituição, o método `produtorConfigs()` da classe `KafkaObjects` ficará dessa forma:

```
public Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();


props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

    return props;
}
```

Vamos cadastrar um item de cardápio com todos os campos preenchidos. Podemos enviar uma mensagem como a seguinte:

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
```

```
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa",
    "descricao": "Uma feijoada onde o céu é o limite"
}'
```

Ao acessarmos o menu "Schema Registry" na interface principal do Redpanda Console, é possível ver o *schema* `restaurantes.cardapio.itens-value` cadastrado.

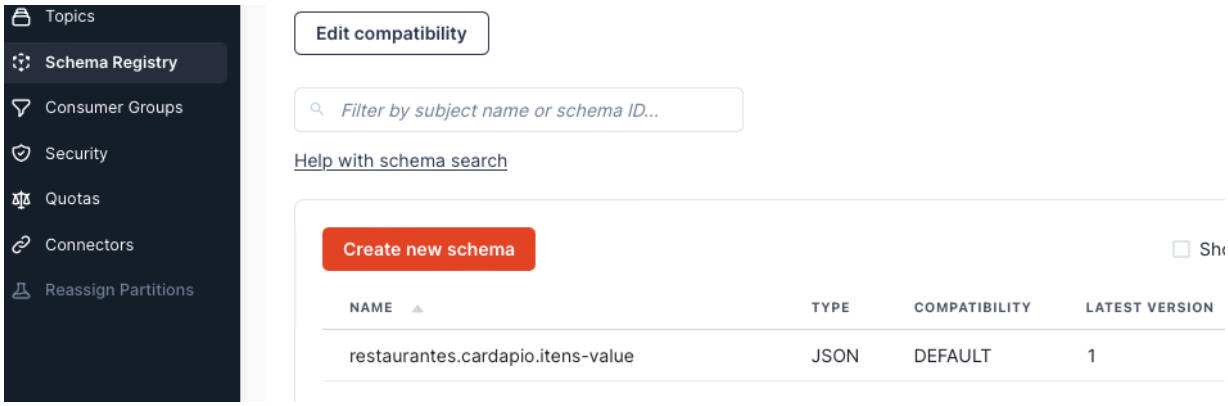


Figura 2.4: Schema cadastrado.

Podemos então enviar novos itens de cardápio para nossa aplicação sem que todos os campos estejam preenchidos. O Schema Registry deveria impedir o recebimento dessas mensagens pelo Kafka — mas não é bem isso que acontece.

Ao enviarmos um item de cardápio sem a descrição do item, usando o seguinte comando:

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa"
}'
```

A mensagem é escrita no tópico sem problemas!

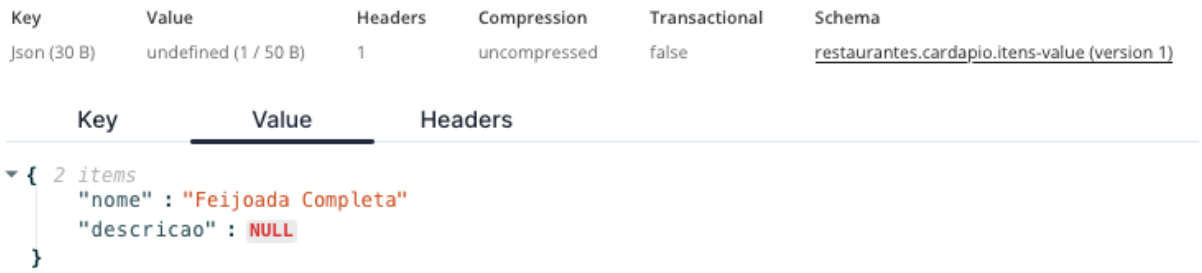


Figura 2.5: Mensagem com campo faltante.

Para entendermos o que ocorreu, vamos analisar o *schema* cadastrado. Ao abri-lo no Redpanda Console, encontramos essa estrutura:

```
{
   "$schema": "http://json-schema.org/draft-
07/schema#",
   "title": "Item Cardapio",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "nome": {
           "oneOf": [
               {
```

```
                    "type": "null",
                    "title": "Not included"
                },
                {
                    "type": "string"
                }
            ]
        },
        "descricao": {
            "oneOf": [
                {
                    "type": "null",
                    "title": "Not included"
                },
                {
                    "type": "string"
                }
            ]
        }
    }
}
```

Repare nos elementos `oneOf`. Eles são necessários para determinar o que pode ser preenchido nos campos onde eles estão inseridos. No caso, eles estão nos dois campos da mensagem (tanto no `nome` quanto na `descricao`). Em ambos os casos, os campos podem ser preenchidos com valores `null` ou com valores `string`. Como o campo `descricao` suporta nulos, o Schema Registry validou a mensagem.

Se quisermos que todos os campos sejam obrigatórios, devemos corrigir esse *schema* cadastrado, substituindo os elementos `oneOf` por elementos mais simples, que determinem apenas um tipo para os campos. Por exemplo, no campo nome:

```
"nome": {
   "type": "string"
}
```

Se fizermos o mesmo com o campo descrição:

```
"descricao": {
   "type": "string"
}
```

Além disso, precisamos informar que os campos são obrigatórios, usando a seguinte configuração:

```
"required": ["nome", "descricao"],
```

O *schema* completo está a seguir:

```
{
   "$schema": "http://json-schema.org/draft-
07/schema#",
   "title": "Item Cardapio",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "nome": {
          "type": "string"
       },
       "descricao": {
```

```
            "type": "string"
        }
    },
    "required": ["nome", "descricao"]
}
```

O elemento `"additionalProperties"` indica que não serão aceitos novos campos na mensagem.

Vamos então:

- remover o *schema* cadastrado pela nossa aplicação;
- substituir pelo *schema* completo já descrito.

Existem dois tipos de exclusão de *schema*: a exclusão *soft*, que pode ser recuperável, mas que não poderá mais ser utilizada por nenhum produtor ou consumidor; e a exclusão *hard*, em que o *schema* é removido fisicamente do Schema Registry, não podendo mais ser acessado. Neste exemplo, usaremos a exclusão *hard*, visto que não queremos mais usar o *schema* cadastrado pelo produtor.

Embora possa parecer estranho à primeira vista, manter *schemas* desabilitados mas ainda cadastrados no Schema Registry pode ser necessário e traz vantagens:

- preservação do histórico e rastreamento de alterações: mantendo mesmo os *schemas* desabilitados cadastrados, temos um histórico de todas as alterações de *schema* realizadas, que podem ser auditadas, se necessário;
- possibilidade de restauração rápida: é possível tornar um *schema* habilitado novamente, caso a desabilitação tenha ocorrido de forma acidental;
- facilidade para testes e experimentação: podemos testar novas versões de um *schema* existente realizando um *soft delete* da sua versão atual. Após a realização dos testes, podemos voltar a utilizá-lo sem que seja necessário um novo cadastro.

O Redpanda Console pode nos ajudar nessa tarefa. Há um botão *Delete subject* presente logo acima do *schema*:

Figura 2.6: Ações que podem ser realizadas no *schema* cadastrado. Entre elas, a de exclusão.

Confirme a exclusão assim que solicitado e clique novamente em *Delete subject*. Assim, a exclusão permanente será feita. Confirme quando necessário.

Para cadastrar o *schema* correto:

- clique no botão "Create new Schema":

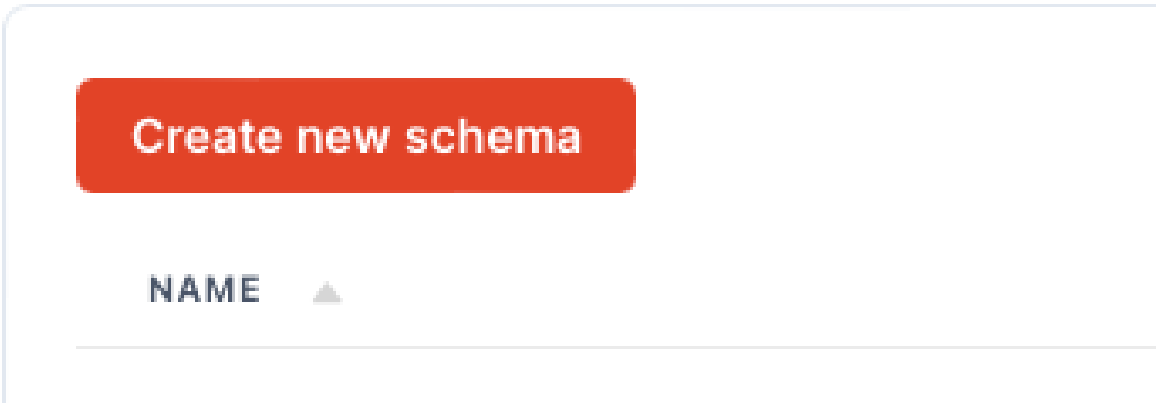


Figura 2.7: Botão de criação de *schema*

- selecione o nome do tópico e na seleção *Key or value*, selecione *Value*. E no *schema definition*, selecione JSON:

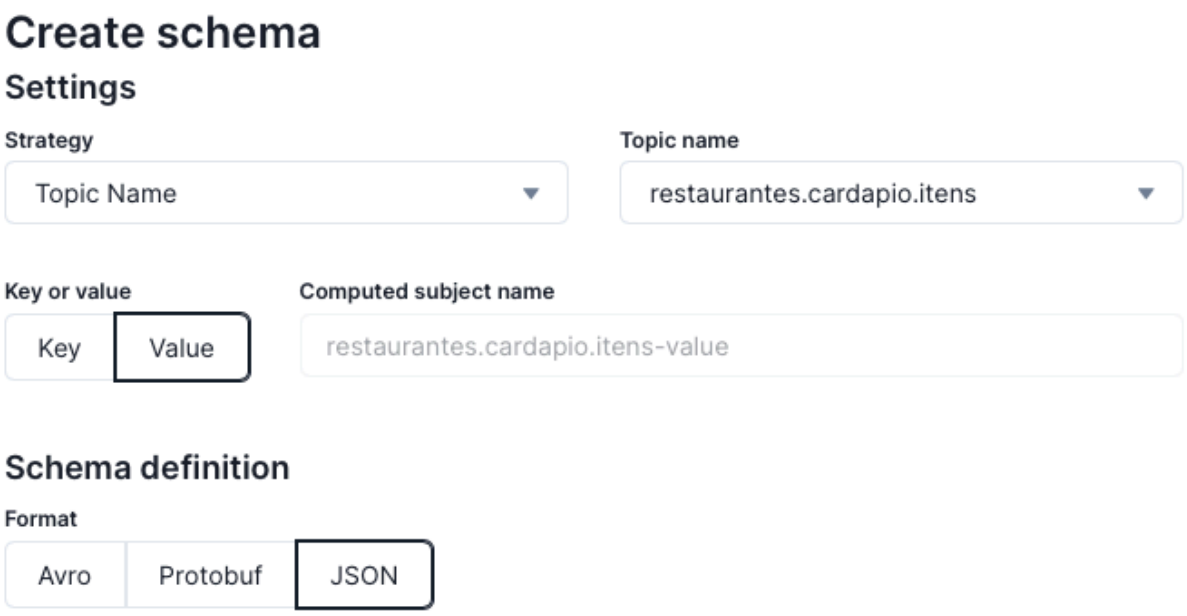


Figura 2.8: Ajustes de cadastro de novo *schema*

Agora, basta inserir o *schema* criado e clicar no botão *Save*.

Pode-se, antes, clicar também no botão *Validate*. Assim é possível verificar se há algum problema antes de cadastrar.

Figura 2.9: Botões para cadastro ou validação de *schema*

Entretanto, ao tentarmos enviar uma mensagem, o *ms-restaurantes* lança a seguinte exceção:

```
2025-02-02T17:52:47.152-03:00 ERROR 14698 --- [ms-
restaurantes] [nio-8080-exec-2] o.a.c.c.C.[.[.[/].
[dispatcherServlet]    : Servlet.service() for servlet
[dispatcherServlet] in context with path [] threw
exception [Request processing failed:
org.apache.kafka.common.errors.SerializationException:
Error serializing JSON message] with root cause

java.io.IOException: Incompatible schema
'{"$schema":"http://json-schema.org/draft/2020-
12/schema#","title":"Identificacao
Item","type":"object","additionalProperties":false,"pro
perties":{"idRestaurante":{"type":"number"},"idItem":
{"type":"number"}},"required":
["idRestaurante","idItem"]}' with refs '[]' of type
'JSON' for schema '{"$schema":"http://json-
schema.org/draft-07/schema#","title":"Identificacao
Item","type":"object","additionalProperties":false,"pro
perties":{"idRestaurante":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"integer"}]},"idItem":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"integer"}]}}}'. Set
latest.compatibility.strict=false to disable this
check. See log file for more details.

//trechos omitidos por brevidade
```

Isso ocorre porque, para o Schema Registry, estamos tentando criar uma mensagem que não é compatível com o *schema* associado ao tópico. Note, na

notificação de erro, que as mensagens podem conter campos nulos (embora não os tenha de fato).

Para que o produtor consiga enviar as mensagens no formato exigido pelo Schema Registry daqui para a frente, precisamos informar que os campos da mensagem são obrigatórios (assim como são no *schema*). Isso é possível inserindo a anotação `@JsonProperty` com o atributo `required = true` da biblioteca *Jackson* nos campos da *record* `ItemCardapio`:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public record ItemCardapio(
        @JsonProperty(required = true)
        String nome,

        @JsonProperty(required = true)
        String descricao) {

}
```

Por último:

- para não correr o risco de o produtor criar uma nova versão do *schema*, precisamos configurá-lo para que ele não consiga escrever novos *schemas*. Isso é possível por meio da configuração a seguir (a ser inserida no método `producerConfigs()` da classe `KafkaObjects`):

```
props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);
```

- para que o produtor trabalhe com a versão mais recente do *schema*, basta adicionar a seguinte configuração:

```
props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);
```

- e para que possamos garantir que o produtor falhe no caso de a mensagem estar fora dos padrões exigidos no *schema* cadastrado, devemos inserir a seguinte configuração:

```
props.put(KafkaJsonSchemaSerializerConfig.FAIL_INVALID_
SCHEMA, "true");
```

As configurações de produtor, portanto, devem ficar assim:

```
public Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();


props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
```

```
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

props.put(KafkaJsonSchemaSerializerConfig.FAIL_INVALID_
SCHEMA, true);
    return props;
}
```

Após essas modificações, se tentarmos cadastrar novamente o item de cardápio sem uma descrição, o *ms-restaurantes* lançará uma exceção, como a seguinte:

```
org.everit.json.schema.ValidationException:
#/descricao: expected type: String, found: Null
    at
org.everit.json.schema.ValidationException.copy(Validat
ionException.java:486) ~[everit-json-schema-
1.14.3.jar:na]
    at
org.everit.json.schema.DefaultValidator.performValidati
on(Validator.java:76) ~[everit-json-schema-
1.14.3.jar:na]
    at
org.everit.json.schema.Schema.validate(Schema.java:152)
~[everit-json-schema-1.14.3.jar:na]
    // trechos omitidos por brevidade
```

2.5 Inserindo um schema para a chave da mensagem

Da mesma maneira que cadastramos um *schema* para a mensagem, podemos fazer o mesmo para sua chave. Felizmente, o processo é bem parecido.

- No método `produtorConfigs()` em `KafkaObjects`, substituir o `JsonSerializer` na configuração do serializador da chave pelo `KafkaJsonSchemaSerializer`;
- Cadastrar o *schema* a seguir para a chave. Note que, assim como o *schema* da mensagem, este *schema* também informa que todos os campos são obrigatórios:

```
{
   "$schema": "http://json-schema.org/draft-
07/schema#",
   "title": "Identificacao Item",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "idRestaurante": {
          "type": "integer"
       },
       "idItem": {
          "type": "integer"
       }
   },
   "required": ["idRestaurante", "idItem"]
}
```

Os *schemas* da mensagem e da chave encontram-se na pasta 'schemas', localizada na raiz do projeto.

- adicionar a anotação `@JsonProperty` nos campos, assim como fizemos nos campos da *record* `ItemCardapio`:

```
import com.fasterxml.jackson.annotation.JsonProperty;

public record IdentificacaoItem(
        @JsonProperty(required = true)
        int idRestaurante,

        @JsonProperty(required = true)
        int idItem) {

}
```

Após o cadastro do *schema* para a chave, ao tentarmos enviar uma mensagem sem um dos campos da chave:

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
--data '{
    "numero": 1,
    "nome": "Feijoada Completa",
    "descricao": "Uma feijoada onde o céu é o limite"
}'
```

A mensagem foi inserida sem problemas, mas o campo não preenchido ficou com valor 0 (zero)!

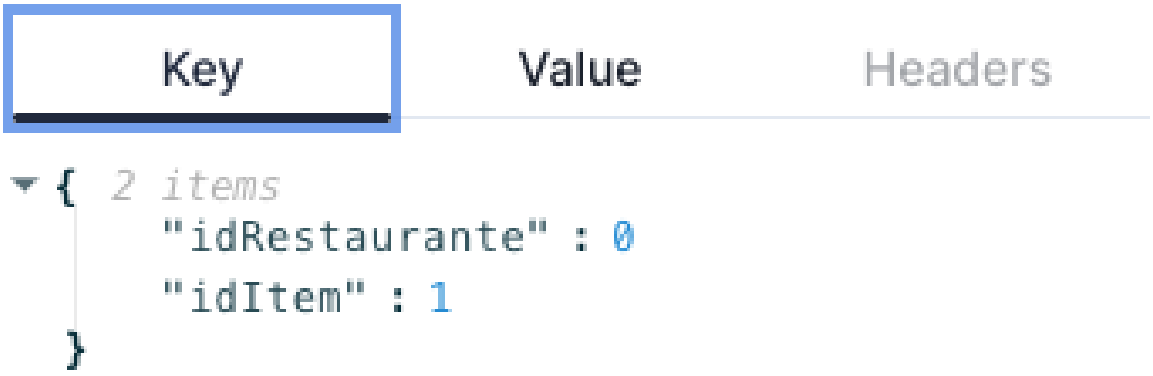


Figura 2.10: id-restaurante com valor zero.

Para entendermos o que aconteceu, vamos observar a classe `ItemCardapioCriadoDTO`. Ela encapsula a mensagem que é enviada via Postman para o *ms-restaurantes*:

```
public class ItemCardapioCriadoDTO {

    @JsonProperty("id-restaurante")
    private int idRestaurante;

    @JsonProperty
    private int numero;

    @JsonProperty
    private String nome;

    @JsonProperty
    private String descricao;

    public ItemCardapioCriado toModel() {
```

```
        return new ItemCardapioCriado(
                new IdentificacaoItem(idRestaurante,
numero),
                new ItemCardapio(nome, descricao));
    }
}
```

Como vimos no início do capítulo, todo tipo primitivo possui um valor padrão, caso não seja atribuído um valor a ele. E foi exatamente isso que ocorreu com os atributos `idRestaurante` e `numero`. Para corrigirmos, vamos substituir o tipo `int` pelo tipo `Integer` em ambos os atributos da classe:

```
@JsonProperty("id-restaurante")
private Integer idRestaurante;

@JsonProperty
private Integer numero;
```

Veremos um problema semelhante a este no capítulo 4.

Agora, se executarmos a chamada novamente, sem o `id-restaurante` na mensagem, o seguinte erro ocorre:

```
org.everit.json.schema.ValidationException: #: required
key [idRestaurante] not found
    at
org.everit.json.schema.ValidationException.copy(Validat
ionException.java:486) ~[everit-json-schema-
1.14.3.jar:na]
```

```
    at
org.everit.json.schema.DefaultValidator.performValidati
on(Validator.java:76) ~[everit-json-schema-
1.14.3.jar:na]
    at
org.everit.json.schema.Schema.validate(Schema.java:152)
~[everit-json-schema-1.14.3.jar:na]

// trechos da stack trace omitidos por brevidade
```

A especificação JSON Schema completa está disponível em: https://json-schema.org/draft-07

2.6 Assincronia dos testes de envio de mensagens e sua correção com Awaitility

Configuramos o Schema Registry para verificar se as mensagens enviadas pelo *ms-restaurantes* estão válidas antes de o tópico recebê-las. Em teoria, já podemos entregar a aplicação em produção para que os restaurantes usem nossa ferramenta.

Mas, ao rodarmos o comando a seguir para gerarmos o pacote que será enviado à produção, executamos todos os testes automatizados. E temos uma surpresa:

```
mvn -f ms-restaurantes/pom.xml package
...
```

```
024-10-13T18:04:33.714-03:00  INFO 3418 --- [ms-
restaurantes] [           main]
o.a.k.c.c.internals.LegacyKafkaConsumer  : [Consumer
clientId=consumer-grupo_de_teste-2,
groupId=grupo_de_teste] Subscribed to topic(s):
restaurantes.cardapio.itens
2024-10-13T18:04:34.264-03:00  INFO 3418 --- [ms-
restaurantes] [           main]
b.c.a.m.MsrestaurantesApplicationTests   : Started
MsrestaurantesApplicationTests in 1.414 seconds
(process running for 28.911)
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 1.537 s -- in
br.com.alura.msrestaurantes.MsrestaurantesApplicationTe
sts
[INFO]
[INFO] Results:
[INFO]
[ERROR] Errors:
[ERROR]
ItensCardapioSenderKafkaTest.lambda$enviarNovoItemCarda
pio$0:38->mensagensDoTopico:46 » RecordDeserialization
Error deserializing key/value for partition
restaurantes.cardapio.itens-1 at offset 0. If needed,
please seek past the record to continue consumption.
[INFO]
[ERROR] Tests run: 3, Failures: 0, Errors: 1, Skipped:
0
[INFO]
[INFO] ------------------------------------------------
------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------
------------------------
[INFO] Total time:  40.242 s
[INFO] Finished at: 2024-10-13T18:04:35-03:00
```

```
[INFO] ------------------------------------------------
------------------------
[ERROR] Failed to execute goal
org.apache.maven.plugins:maven-surefire-
plugin:3.2.5:test (default-test) on project ms-
restaurantes:
[ERROR]
[ERROR] Please refer to
/Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livr
o/branch/rodrigo-vieira-pinto/codigo-fonte/ms-
restaurantes/target/surefire-reports for the individual
test results.
```

O teste de envio de mensagens está falhando! Esse teste é representado pela classe `ItensCardapioSenderKafkaTest`. Essa classe executa basicamente os seguintes passos:

- por meio de uma instância de `ItemCardapioSenderKafka` é enviado um `ItemCardapio` para o tópico;
- o teste recebe uma instância de `KafkaConsumer` injetada pelo Spring e este consome a mensagem recém-enviada;
- por último, o teste verifica se o `ItemCardapio` recebido pela instância de `KafkaConsumer` é o que foi enviado no primeiro passo.

Esse tipo de teste é considerado um teste *assíncrono*, visto que não podemos verificar se a mensagem chegou ao tópico logo em seguida ao seu

envio, sob risco de tornar o teste frágil. Às vezes, o teste informará que passou; outras vezes, não.

Uma possível abordagem nesse tipo de teste seria, de tempos em tempos, verificarmos se a mensagem foi enviada ao tópico dentro de um período máximo de tempo. Se dentro deste período a mensagem estiver presente, o teste passa. Caso contrário, ele falha.

Precisamos, então, trabalhar com ferramentas que nos auxiliem com esse tipo de teste, como a **Awaitility**. Com ela, podemos lidar com a incerteza sobre quando a mensagem estará no tópico de maneira elegante.

> Confira mais informações sobre essa biblioteca no site http://www.awaitility.org e no livro *Kafka Connect e Kafka Streams: Integre suas aplicações ao Kafka usando as melhores práticas* (2024) deste mesmo autor.

Ela já está presente no `pom.xml` do *ms-restaurantes*:

```
<dependency>
   <groupId>org.awaitility</groupId>
   <artifactId>awaitility</artifactId>
   <scope>test</scope>
</dependency>
```

Utilizando-se de uma DSL (*Domain Specific Language*), podemos criar uma sentença como a seguinte:

- aguarde por no máximo 10 segundos para que se faça as verificações do teste.

Essa sentença está presente em `ItensCardapioSenderKafkaTest` da seguinte forma:

```
private void esperarUmTempoAte(ThrowingRunnable
runnable){
    await()
        .atMost(Duration.ofSeconds(10))
        .untilAsserted(runnable);
}
```

Esse método é utilizado neste teste:

```
@Test
void enviarNovoItemCardapio(){
    var itemCardapioCriado = umItemDeCardapioCriado();

baseItensCardapioKafka.adicionar(itemCardapioCriado);

    esperarUmTempoAte(() ->
            assertThat(mensagensDoTopico())

.containsEntry(itemCardapioCriado.identificacaoItem(),

itemCardapioCriado.itemCardapio()));
}
```

O método `mensagensDoTopico()` retorna todas as mensagens lidas. Ele faz uma verificação por segundo no tópico em busca da mensagem. Como temos 10 segundos de tempo limite, serão realizadas cerca de 10 verificações no tópico. Se a mensagem não chegar nesse tempo, o teste falhará:

```
private Map<IdentificacaoItem, ItemCardapio>
mensagensDoTopico(){
    Map<IdentificacaoItem, ItemCardapio> eventos = new
HashMap<>();


consumidor.poll(Duration.ofSeconds(1)).forEach(consumer
Record ->
        eventos.put(consumerRecord.key(),
consumerRecord.value())
    );

    return eventos;
}
```

A instância de `KafkaConsumer` está configurada na classe `KafkaObjectsParaTestes` (presente no mesmo pacote que `ItensCardapioSenderKafkaTest`):

```
package br.com.alura.msrestaurantes.infra.kafka;

import
br.com.alura.msrestaurantes.dominio.IdentificacaoItem;
import
br.com.alura.msrestaurantes.dominio.ItemCardapio;
import
```

```
org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.kafka.support.serializer.JsonDeseri
alizer;

import java.util.Map;

import static java.util.Collections.singletonList;

@Configuration
class KafkaObjectsParaTestes {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.consumer.key-deserializer}")
    private String keyDeserializer;

    @Value("${spring.kafka.consumer.value-
deserializer}")
    private String valueDeserializer;

    @Value("${spring.kafka.consumer.group-id}")
    private String grupo;

    @Value("${spring.kafka.consumer.auto-offset-
reset}")
    private String autoOffsetReset;

    @Value("${topico.nome}")
```

```
    private String nomeDoTopico;

    @Bean
    KafkaConsumer<IdentificacaoItem, ItemCardapio>
consumidorDeMensagens(){
        KafkaConsumer<IdentificacaoItem, ItemCardapio>
consumidor =
                new KafkaConsumer<>
(configsDoConsumidor());

consumidor.subscribe(singletonList(nomeDoTopico));
        return consumidor;
    }

    public Map<String, Object> configsDoConsumidor() {
        return Map.of(

ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaServerUrl
                ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
keyDeserializer
                ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
valueDeserializer
                , ConsumerConfig.GROUP_ID_CONFIG, grupo
                ,
ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
                , JsonDeserializer.TRUSTED_PACKAGES,
"br.com.alura.msrestaurantes.dominio"
        );
    }

}
```

O `KafkaConsumer` é uma classe da biblioteca `spring-kafka` que nos permite consultar as mensagens provenientes de um tópico. Podemos, inclusive, configurar de quanto em quanto tempo ela deverá acessar o tópico em busca de mensagens, entre outros ajustes.

> Vale a pena consultar o livro *Apache Kafka e Spring Boot: Comunicação assíncrona entre microsserviços* de Eduardo Santana (2022) caso você tenha dúvidas sobre como funciona a classe `KafkaConsumer`.

Repare que, no método `configsDoConsumidor()`, não há nenhuma configuração que faça menção ao Schema Registry, o que faz sentido visto que, quando esses testes foram criados, ele ainda não fazia parte da nossa solução.

De forma semelhante ao que fizemos nas configurações do produtor, devemos também adicionar a URL do Schema Registry no nosso consumidor de testes, acrescentando as linhas a seguir:

- nos atributos de `KafkaObjectsParaTestes`:

```
@Value("${spring.kafka.schema.registry.url}")
private String schemaRegistryUrl;
```

- e como configuração no método `configsDoConsumidor()`:

```
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, schemaRegistryUrl
```

- devemos também informar quais classes do consumidor serão utilizadas na desserialização da chave e do valor da mensagem. Adicionamos as configurações a seguir:

```
KafkaJsonSchemaDeserializerConfig.JSON_KEY_TYPE, IdentificacaoItem.class.getName()
KafkaJsonSchemaDeserializerConfig.JSON_VALUE_TYPE, ItemCardapio.class.getName()
```

- também deixamos de usar a configuração `JsonDeserializer.TRUSTED_PACKAGES`, já que fizemos as configurações anteriores.
- por último, no arquivo `application-test.yml`, devemos substituir o desserializador utilizado na chave e no valor pelo `io.confluent.kafka.serializers.json.KafkaJsonSchemaDeserializer`:

```
key-deserializer: io.confluent.kafka.serializers.json.KafkaJsonSchemaDeserializer
value-deserializer: io.confluent.kafka.serializers.json.KafkaJsonSchemaDeserializer
```

Com essas alterações, o método `configsDoConsumidor()` ficará como a seguir:

```
public Map<String, Object> configsDoConsumidor() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
keyDeserializer
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
valueDeserializer
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
            ,
KafkaJsonSchemaDeserializerConfig.JSON_KEY_TYPE,
IdentificacaoItem.class.getName()
            ,
KafkaJsonSchemaDeserializerConfig.JSON_VALUE_TYPE,
ItemCardapio.class.getName()
    );
}
```

E o arquivo `application-test.yml` ficará assim:

```
spring:

  kafka:
    consumer:
      group-id: grupo_de_teste
```

```
      auto-offset-reset: earliest
      key-deserializer:
io.confluent.kafka.serializers.json.KafkaJsonSchemaDese
rializer
      value-deserializer:
io.confluent.kafka.serializers.json.KafkaJsonSchemaDese
rializer
```

Com essas alterações, os testes do *ms-restaurantes* voltaram a passar sem problemas, e o pacote será gerado normalmente:

```
mvn -f ms-restaurantes/pom.xml package
...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...
[INFO] ------------------------------------------------
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
------------------------
[INFO] Total time:  01:03 min
[INFO] Finished at: 2024-10-13T19:16:32-03:00
[INFO] ------------------------------------------------
------------------------
```

Diversos trechos do *log* foram omitidos por brevidade. E com isso encerramos o trabalho necessário para o produtor enviar mensagens corretamente para o tópico Kafka.

O @EmbeddedKafka

No início da classe `ItensCardapioSenderTest`, é possível encontrar a anotação `@EmbeddedKafka`:

```
@ActiveProfiles("test")
@SpringBootTest
@EmbeddedKafka
class ItensCardapioSenderKafkaTest {
```

Essa anotação nos permite testar a integração de nossa aplicação com o Kafka sem que a aplicação precise se conectar a uma instância real da ferramenta. A aplicação vai se conectar a uma instância em memória e enviar mensagens a ela.

Se rodarmos esse teste, é possível ver que nenhuma mensagem chegará no tópico `restaurantes.cardapio.itens`. Porém, pode-se desejar executar um teste integrado com nossa aplicação acessando o Kafka real e enviando mensagens para lá. Nesse caso, tudo o que teremos que fazer é remover ou simplesmente comentar o `@EmbeddedKafka`, como a seguir. As mensagens serão enviadas ao Kafka sem falhas. Confira:

```
@ActiveProfiles("test")
@SpringBootTest
// @EmbeddedKafka
class ItensCardapioSenderKafkaTest {
```

Para utilizar esse trecho em nosso projeto, precisamos apenas inserir a biblioteca `spring-kafka-test` no nosso `pom.xml` :

```
<dependency>
   <groupId>org.springframework.kafka</groupId>
   <artifactId>spring-kafka-test</artifactId>
   <scope>test</scope>
</dependency>
```

Essa anotação suporta diversas configurações, sendo um recurso interessante para a realização de testes unitários de produtores e consumidores de mensagens. Consulte os *links* https://docs.spring.io/spring-kafka/reference/testing.html e https://docs.spring.io/spring-kafka/api/org/springframework/kafka/test/context/EmbeddedKafka.html para mais informações.

2.7 Mas e o consumidor?

Ao subir o *ms-pedidos*, veremos que ele, após a inserção do Schema Registry, deixa de consumir mensagens e lança exceções que explicam os motivos do problema:

```
java.lang.IllegalStateException: This error handler
cannot process 'SerializationException's directly;
please consider configuring an
'ErrorHandlingDeserializer' in the value and/or key
deserializer
    at
org.springframework.kafka.listener.DefaultErrorHandler.
handleOtherException(DefaultErrorHandler.java:192) ~
[spring-kafka-3.2.4.jar:3.2.4]
    at
org.springframework.kafka.listener.KafkaMessageListener
Container$ListenerConsumer.handleConsumerException(Kafk
aMessageListenerContainer.java:1925) ~[spring-kafka-
3.2.4.jar:3.2.4]

...

Caused by:
org.apache.kafka.common.errors.RecordDeserializationExc
eption: Error deserializing key/value for partition
restaurantes.cardapio.itens-0 at offset 0. If needed,
please seek past the record to continue consumption.
    at
org.apache.kafka.clients.consumer.internals.CompletedFe
tch.parseRecord(CompletedFetch.java:331) ~[kafka-
clients-3.7.1.jar:na]
    at
org.apache.kafka.clients.consumer.internals.CompletedFe
tch.fetchRecords(CompletedFetch.java:283) ~[kafka-
clients-3.7.1.jar:na]
    at
org.apache.kafka.clients.consumer.internals.FetchCollec
tor.fetchRecords(FetchCollector.java:168) ~[kafka-
clients-3.7.1.jar:na]

...
```

```
Caused by:
org.apache.kafka.common.errors.SerializationException:
Can't deserialize data  from topic
[restaurantes.cardapio.itens]
    at
org.springframework.kafka.support.serializer.JsonDeseri
alizer.deserialize(JsonDeserializer.java:587) ~[spring-
kafka-3.2.4.jar:3.2.4]
    at
org.apache.kafka.common.serialization.Deserializer.dese
rialize(Deserializer.java:73) ~[kafka-clients-
3.7.1.jar:na]
    at
org.apache.kafka.clients.consumer.internals.CompletedFe
tch.parseRecord(CompletedFetch.java:319) ~[kafka-
clients-3.7.1.jar:na]
    ... 14 common frames omitted
Caused by: java.io.CharConversionException: Invalid
UTF-32 character 0x27a2269 (above 0x0010ffff) at char
#1, byte #7)
    at
com.fasterxml.jackson.core.io.UTF32Reader.reportInvalid
(UTF32Reader.java:193) ~[jackson-core-
2.17.2.jar:2.17.2]
    at
com.fasterxml.jackson.core.io.UTF32Reader.read(UTF32Rea
der.java:159) ~[jackson-core-2.17.2.jar:2.17.2]
    at
com.fasterxml.jackson.core.json.ReaderBasedJsonParser._
loadMore(ReaderBasedJsonParser.java:250) ~[jackson-
core-2.17.2.jar:
```

Assim como em códigos anteriores, a mensagem teve trechos omitidos por brevidade.

Nos próximos capítulos, veremos as correções a serem aplicadas neste microsserviço.

2.8 Diferenças entre o JsonSerializer e o KafkaJsonSchemaSerializer

Como já vimos neste capítulo, utilizando o `KafkaJsonSchemaSerializer` como serializador de mensagens, o produtor se integrará ao Schema Registry, verificando, portanto, se a mensagem está de acordo com o *schema* cadastrado para o tópico. É um comportamento que o `JsonSerializer` não possui, visto que nele a serialização é feita com base nos tipos presentes no *header* da mensagem.

Mas, ao usarmos o `KafkaJsonSchemaSerializer`, os tipos não são enviados. Vide figura a seguir:

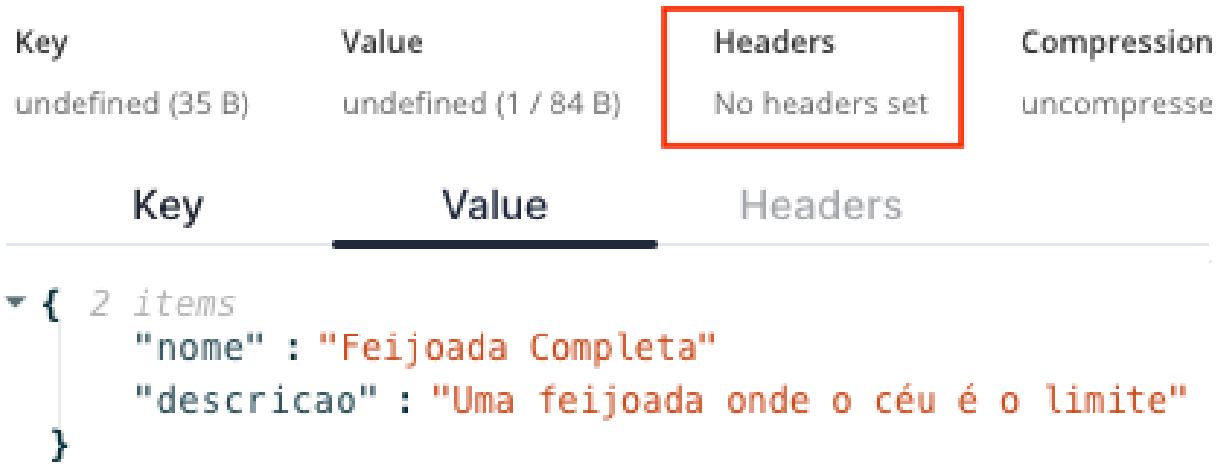


Figura 2.11: Com o novo serializer, a mensagem não possui headers

Isso ocorre porque, no caso do `KafkaJsonSchemaSerializer`, o foco não é garantir o polimorfismo via *headers* como o `_TypeId_`, mas sim garantir que as mensagens estejam em conformidade com o *schema* registrado no Schema Registry. Ou seja, ele se preocupa mais com a validade do *schema* do que com o mapeamento direto de tipos Java via *headers*.

Resumindo: enquanto o `JsonSerializer` se preocupa em oferecer garantias no formato da mensagem por meio dos tipos usados na serialização, o `KafkaJsonSchemaSerializer` usa o Schema Registry para essa tarefa.

2.9 Resumo

Neste capítulo, vimos como o Schema Registry nos ajuda a controlar as mensagens enviadas para um tópico e como configurar um produtor de mensagens JSON para trabalhar com ele.

No próximo capítulo, abordaremos mais detalhes sobre o Schema Registry. Vamos entender melhor como manipular *schemas*, além de entender o funcionamento do JSON Schema.

Capítulo 3

Explorando a API do Schema Registry: Endpoints, Subjects e Schemas

Assim como o Kafka, o Schema Registry não possui uma interface gráfica oficial para que possamos interagir com ele. Em vez disso, possui uma API REST que nos permite realizar diversas operações.

Além disso, por mais que o Redpanda Console seja uma interface gráfica interessante para realizar operações manuais no Kafka e no Schema Registry, certamente ele não é a melhor escolha quando precisamos automatizar essas operações. Enquanto interfaces gráficas exigem cliques e entradas manuais, APIs possibilitam a execução de operações de forma programática e escalável. Logo, a Schema Registry API é o recurso mais adequado para esses casos.

Veremos alguns exemplos dessa API usando os *schemas* e operações vistas no capítulo anterior.

Os códigos presentes neste capítulo encontram-se na *branch* `cap-3` do projeto no GitHub. Nesta *branch*, foram introduzidos *scripts* de cadastro dos *schemas* vistos no capítulo anterior, facilitando a subida do ambiente para testar a solução. Tanto esses *scripts* quanto os *schemas* estão na pasta `schemas`, localizada na raiz do projeto. O *script* `infra-up.sh` executa esses *scripts*.

3.1 O subject e seus schemas

Um *subject* é um identificador lógico que agrupa e versiona *schemas* que representam a estrutura dos dados de um tópico Kafka específico. Em resumo, ele organiza e controla o versionamento dos *schemas*. Cada *subject* pode conter várias versões de *schemas* ao longo do tempo. Já um *schema* é a estrutura de dados que define como uma mensagem deve ser formatada. Ele especifica os tipos de dados e as regras de cada campo.

Como exemplo, o tópico `restaurantes.cardapio.itens`, em que trabalhamos até o momento, possui dois *subjects*: um para a chave da mensagem e outro para o valor:

- `restaurantes.cardapio.itens-key`
- `restaurantes.cardapio.itens-value`

E eles possuem respectivamente os seguintes *schemas*:

```
{
   "$schema": "http://json-schema.org/draft/schema#",
   "title": "Identificacao Item",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "idRestaurante": {
          "type": "number"
       },
       "idItem": {
          "type": "number"
       }
   },
   "required": ["idRestaurante", "idItem"]
}

{
   "$schema": "http://json-schema.org/draft/schema#",
   "title": "Item Cardapio",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "nome": {
          "type": "string"
       },
       "descricao": {
          "type": "string"
       }
   },
```

```
   "required": ["nome", "descricao"]
}
```

Durante o capítulo 1, realizamos as seguintes operações:

- cadastro de *schemas* da chave e do valor da mensagem, com possibilidade de validação de ambos antes do cadastro;
- atualização de ambos os schemas;
- listagem dos *schemas* previamente cadastrados.

Também é possível verificar no Redpanda Console outras informações referentes ao Schema Registry, na tela principal de visualização do Schema Registry:

- o *Schema ID* : um identificador único para o *schema* dentro da aplicação.
- a versão atual do *schema*.
- o tipo do *schema* cadastrado (JSON, AVRO, Protobuf).
- a versão mais recente de cada *schema* .
- busca por nome do *schema* ou seu *Schema ID*.
- remoção de *schemas*.

A seguir, veja como aparece na tela, segundo a versão da ferramenta que estamos usando:

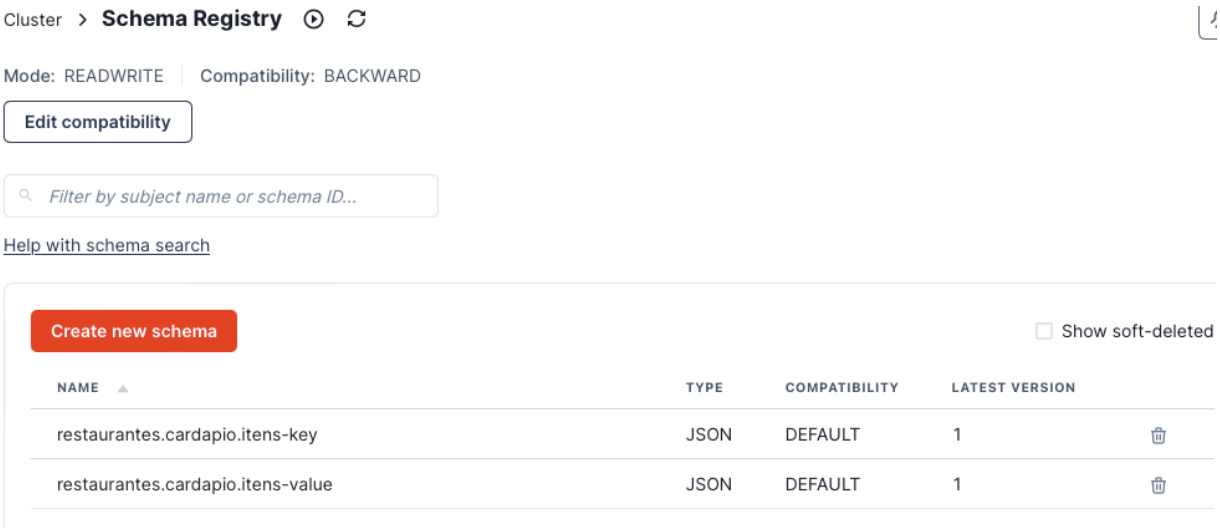


Figura 3.1: Interface principal do Schema Registry oferecida pelo Redpanda Console.

Todas essas operações também podem ser realizadas via Schema Registry API e, neste capítulo, veremos como usá-la. A atualização e a validação de *schemas* será vista no capítulo 7.

3.2 Cadastrando schemas

Roy Fielding, em sua tese sobre REST (2000), nos indicou que deveríamos fazer uso dos recursos do protocolo HTTP de forma a simplificar cada vez mais o entendimento de APIs, tornando-as mais intuitivas. A partir desse trabalho, diversas APIs em todo o mundo foram produzidas seguindo essas ideias. E com a API REST do Schema Registry não é diferente.

Por exemplo, para cadastrar um novo *schema*, podemos usar o seguinte *endpoint*:

- `POST /subjects/<<subject>>/versions`

onde `<<subject>>` é o nome do *schema* a ser cadastrado. É necessário enviar o *schema* a ser cadastrado no corpo da mensagem.

A seguir, veja um exemplo de sua utilização, usando o comando cURL para cadastrar o *schema* da chave do tópico `restaurantes.cardapio.itens-key` :

```
curl -X POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-key/versions -H 'Content-Type:
application/vnd.schemaregistry.v1+json' -d
@restaurantes.cardapio.itens-key.js
```

O conteúdo do arquivo `restaurantes.cardapio.itens-key.js` é o mesmo já visto no início deste capítulo:

```
{
   "$schema": "http://json-schema.org/draft/schema#",
   "title": "Identificacao Item",
   "type": "object",
   "additionalProperties": false,
   "properties": {
       "idRestaurante": {
          "type": "number"
       },
       "idItem": {
          "type": "number"
       }
```

```
    },
    "required": ["idRestaurante", "idItem"]
}
```

Por se tratar do JSON que utilizamos no Redpanda Console para cadastrar esse *schema*, teoricamente devemos usá-lo também com a API REST do Schema Registry. No entanto, ao executarmos o cURL descrito, obtemos o seguinte erro:

```
rodrigovieirapinto@MBP-de-Rodrigo schemas % curl -X
POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-key/versions -H 'Content-Type:
application/vnd.schemaregistry.v1+json' -d
@restaurantes.c
ardapio.itens-key.js
{"error_code":42201,"message":"Empty schema"}%
```

Esse problema ocorre devido ao formato exigido pela API do Schema Registry para o corpo da mensagem. A seguir, temos algumas instruções necessárias:

- o corpo deve estar contido em um elemento JSON chamado de `schema`. Pelo fato de esse elemento não existir na mensagem, a resposta do Schema Registry foi o erro já mencionado;
- por padrão, os *schemas* suportados são do tipo AVRO. Caso seja necessário cadastrar um *schema* em outro formato, é necessário informar isso explicitamente na mensagem.

Dessa forma, devemos realizar algumas alterações no arquivo `restaurantes.cardapio.itens-key.js`, seguindo essas observações. Após as alterações, o arquivo ficará como o seguinte:

```
{
   "schemaType": "JSON",
   "schema": "{
      \"$schema\": \"http://json-
schema.org/draft/schema#\",
      \"title\": \"Identificacao Item\",
      \"type\": \"object\",
      \"additionalProperties\": false,
      \"properties\": {
         \"idRestaurante\": {
            \"type\": \"number\"
         },
         \"idItem\": {
             \"type\": \"number\"
         }
      },
      \"required\": [\"idRestaurante\", \"idItem\"]
   }"
}
```

Repare que o novo elemento `schema` contém todo o conteúdo anterior do arquivo, mas formatado como se fosse uma única `String`. Isso se deve ao valor suportado neste campo. Ele não pode ser um objeto JSON, mas apenas `String`.

Após esses ajustes, é possível cadastrar este *schema* sem problemas:

```
rodrigovieirapinto@MBP-de-Rodrigo schemas % curl -X
POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-key/versions -H 'Content-Type:
application/vnd.schemaregistry.v1+json' -d
@restaurantes.cardapio.itens-key.js
{"id":1}
```

A resposta desse *endpoint* é o **Schema ID**, um identificador único para o *schema* cadastrado no Schema Registry. Como não havia *schemas* cadastrados anteriormente, esse *schema* recebeu o identificador **1**.

Podemos aplicar os conhecimentos adquiridos no cadastro do *schema* da chave da mensagem para cadastrar o *schema* do valor da mensagem.

Após as modificações, o `restaurantes.cardapio.itens-value.js` deve ficar assim:

```
{
   "schemaType": "JSON",
   "schema": "{
      \"$schema\": \"http://json-
schema.org/draft/schema#\",
      \"title\": \"Item Cardapio\",
      \"type\": \"object\",
      \"additionalProperties\": false,
      \"properties\": {
```

```
            \"nome\": {
               \"type\": \"string\"
            },
            \"descricao\": {
               \"type\": \"string\"
            }
         },
         \"required\": [\"nome\", \"descricao\"]
   }"
}
```

Após rodarmos o seguinte cURL, recebemos o Schema ID do novo *schema* cadastrado como resposta:

```
rodrigovieirapinto@MBP-de-Rodrigo schemas % curl -X
POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions -H 'Content-Type:
application/vnd.schemaregistry.v1+json' -
d@restaurantes.cardapio.itens-value.js
{"id":2}%
```

Esses comandos cURL estão encapsulados nos scripts a seguir, facilitando sua execução no Schema Registry no Schema Registry:

- `restaurantes.cardapio.itens-key-cadastrar.sh`
- `restaurantes.cardapio.itens-value-cadastrar.sh`

De fato, ao visualizarmos o *schema* recém-cadastrado no Redpanda Console, podemos ver que

ele recebeu o Schema ID informado pela API do Schema Registry:

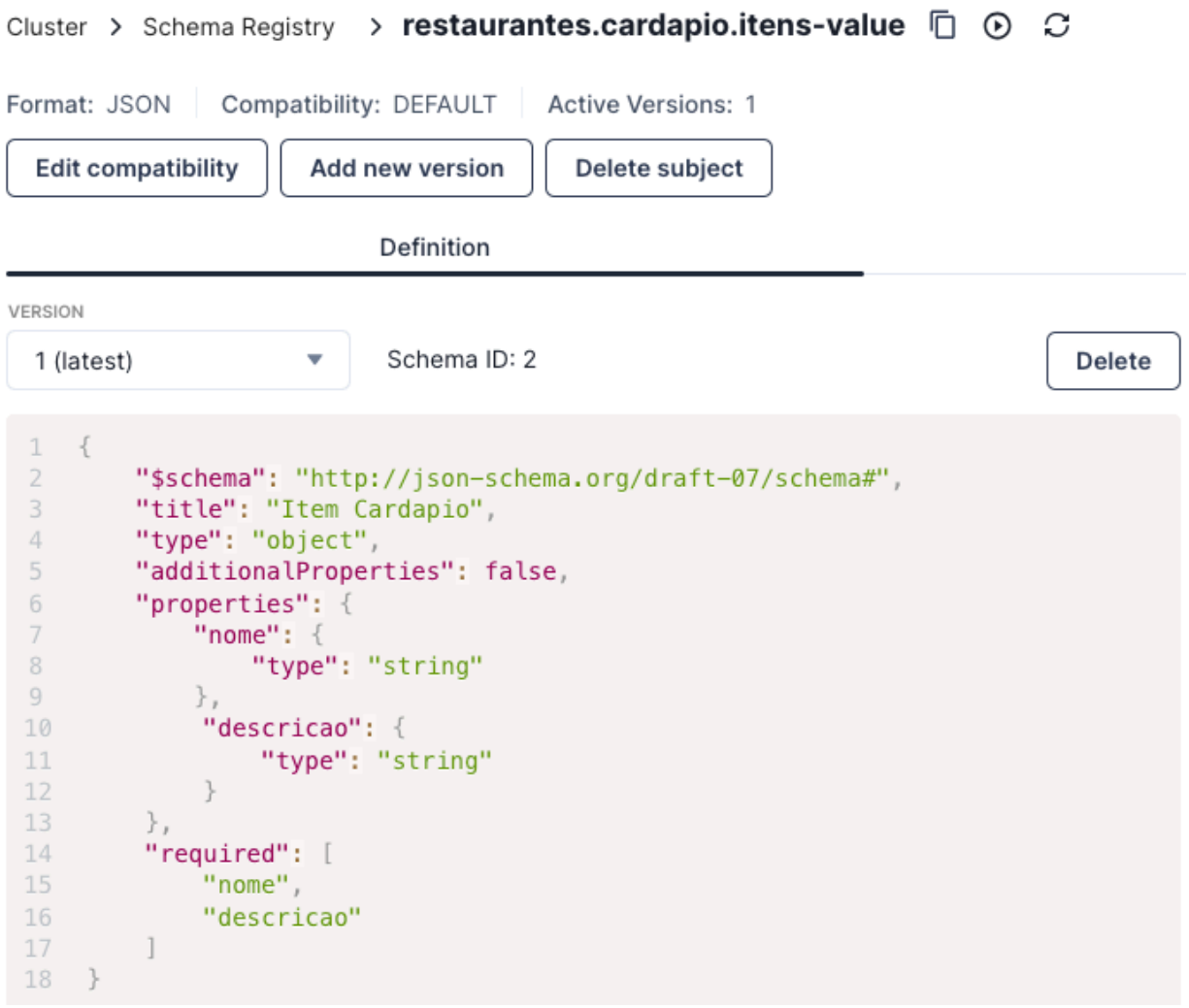


Figura 3.2: Schema cadastrado, juntamente com seu respectivo Schema ID.

Não confundir Schema ID com a versão do schema *(representado pelo campo* version *presente na figura anterior)!* É possível trabalharmos com diversas versões de *schema*, seja para a chave da mensagem, seja para o valor da mensagem. O conceito de versões do *schema* é o mesmo aplicado nas versões que uma API REST pode ter. Ou seja: cada vez que se atualiza o *schema*, uma nova versão dele é gerada.

3.3 Listando schemas

Podemos listar os dois *schemas* cadastrados anteriormente usando o cURL:

```
curl -X GET http://localhost:8081/schemas
[{"subject":"restaurantes.cardapio.itens-
key","version":1,"id":1,"schemaType":"JSON","schema":"
{\"$schema\":\"http://json-schema.org/draft/2020-
12/schema#\",\"title\":\"Identificacao
Item\",\"type\":\"object\",\"additionalProperties\":fal
se,\"properties\":{\"idRestaurante\":
{\"type\":\"number\"},\"idItem\":
{\"type\":\"number\"}},\"required\":
[\"idRestaurante\",\"idItem\"]}"},
{"subject":"restaurantes.cardapio.itens-
value","version":1,"id":2,"schemaType":"JSON","schema":
"{\"$schema\":\"http://json-schema.org/draft/2020-
12/schema#\",\"title\":\"Item
```

```
Cardapio\",\"type\":\"object\",\"additionalProperties\"
:false,\"properties\":{\"nome\":
{\"type\":\"string\"},\"descricao\":
{\"type\":\"string\"}},\"required\":
[\"nome\",\"descricao\"]}"}]%
```


Se quisermos obter informações sobre apenas um deles, podemos informar o Schema ID na URL. Nos exemplos a seguir, buscamos primeiro o *schema* da chave e depois o *schema* do valor previamente cadastrado:

```
rodrigovieirapinto@MBP-de-Rodrigo schemas % curl -X GET
http://localhost:8081/schemas/ids/1
{"schemaType":"JSON","schema":"
{\"$schema\":\"http://json-schema.org/draft/2020-
12/schema#\",\"title\":\"Identificacao
Item\",\"type\":\"object\",\"additionalProperties\":fal
se,\"properties\":{\"idRestaurante\":
{\"type\":\"number\"},\"idItem\":
{\"type\":\"number\"}},\"required\":
[\"idRestaurante\",\"idItem\"]}"}%

rodrigovieirapinto@MBP-de-Rodrigo schemas % curl -X GET
http://localhost:8081/schemas/ids/2
{"schemaType":"JSON","schema":"
{\"$schema\":\"http://json-schema.org/draft/2020-
12/schema#\",\"title\":\"Item
Cardapio\",\"type\":\"object\",\"additionalProperties\"
:false,\"properties\":{\"nome\":
{\"type\":\"string\"},\"descricao\":
{\"type\":\"string\"}},\"required\":
[\"nome\",\"descricao\"]}"}%
```

3.4 Outras buscas

Para obter o *schema*, seu respectivo Schema ID e outras informações de um *subject*, usamos o seguinte *endpoint*:

```
GET /subjects/<<subject>>/versions/latest
```

Exemplo: ao usarmos o `restaurantes.cardapio.itens-key` como *subject*, temos a seguinte resposta:

```
curl -X GET
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-key/versions/latest
{"subject":"restaurantes.cardapio.itens-
key","version":1,"id":1,"schemaType":"JSON","schema":"
{\"$schema\":\"http://json-schema.org/2020-
12/schema#\",\"title\":\"Identificacao
Item\",\"type\":\"object\",\"additionalProperties\":fal
se,\"properties\":{\"idRestaurante\":
{\"type\":\"number\"},\"idItem\":
{\"type\":\"number\"}},\"required\":
[\"idRestaurante\",\"idItem\"]}"}%
```

Para buscar todos os *subjects*, temos:

```
GET /subjects
```

Exemplo:

```
curl -X GET http://localhost:8081/subjects
["restaurantes.cardapio.itens-
key","restaurantes.cardapio.itens-value"]%
```

3.5 Remoção soft e hard de schemas

Para realizarmos um *soft delete* de um *schema* específico, podemos usar sua versão como critério de busca no *endpoint*:

```
DELETE /subjects/<<subject>>/versions/<<versão>>
```

Como exemplo de uso desse *endpoint*, podemos utilizar o *schema* cadastrado para o *subject* `restaurantes.cardapio.itens-key`. Note que, se tudo ocorrer sem problemas, será retornado o Schema ID do *schema* como resposta:

```
curl -X DELETE
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-key/versions/1
1%
```

Para remover um *schema* definitivamente, precisamos apenas inserir um parâmetro no *endpoint* usado para *soft delete*:

```
DELETE /subjects/<<subject>>/versions/<<versão>>?
permanent=true
```

> **IMPORTANTE:** para que esse *endpoint* funcione, é necessário que um *soft delete* tenha sido realizado previamente no *schema*. Caso contrário, o *endpoint* não funcionará.

Vejamos um exemplo de uso do *endpoint* agora usando o *schema* de valor da mensagem. Como o *soft delete* não foi executado antes, o *endpoint* informa que a versão que se tenta remover não foi previamente desabilitada:

```
curl -X DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-value/versions/1?permanent=true'
{"error_code":40407,"message":"Subject
'restaurantes.cardapio.itens-value' Version 1 was not
deleted first before being permanently deleted"}%
```

Uma vez que o *soft delete* é feito, o *hard delete* é aceito sem problemas. Assim como no *soft delete*, o *hard delete* também retorna o Schema ID do *schema* manipulado:

```
rodrigovieirapinto@MBP-de-Rodrigo docker % curl -X
DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-value/versions/1'
1%
rodrigovieirapinto@MBP-de-Rodrigo docker % curl -X
DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-value/versions/1?permanent=true'
1%
```

Se quisermos realizar um *hard delete* do *schema* da chave das mensagens:

```
rodrigovieirapinto@MBP-de-Rodrigo docker % curl -X
DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-key/versions/1?permanent=true'
```

3.6 Remoção soft e hard de subjects

As mesmas regras relacionadas a *soft* e *hard delete* implementadas na remoção de *schemas* aplicam-se também à remoção de *subjects*. E os *endpoints* são semelhantes.

Para demonstrar a remoção de *subjects*, vamos realizar a sua reinstalação, usando os *scripts* presentes na pasta `schemas` :

```
cd ..
cd schemas
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./restaurantes.cardapio.itens-key-cadastrar.sh
{"id":3}%
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./restaurantes.cardapio.itens-value-cadastrar.sh
{"id":4}%
```

Para realizar um *soft delete*, usamos o seguinte *endpoint*:

```
DELETE /subjects/<<subject>>
```

Tomando os *subjects* cadastrados como exemplo, temos:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-key'
[1]%

rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-value'
[1]%
```

Também podemos buscar todos os *subjects* que foram desabilitados. Para isso, basta usar o endpoint `GET /subjects?deleted=true`.

Veja um exemplo de uso. Nesse caso, serão retornados não somente o `subject` do valor da mensagem, mas também o *subject* da chave da mensagem:

```
rodrigovieirapinto@MBP-de-Rodrigo docker % curl -X GET
'http://localhost:8081/subjects?deleted=true'
["restaurantes.cardapio.itens-
key","restaurantes.cardapio.itens-value"]%
```

E para fazermos um *hard delete*:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
```

```
tens-key?permanent=true'
[1]%

rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X DELETE
'http://localhost:8081/subjects/restaurantes.cardapio.i
tens-value?permanent=true'
[1]%
```

A lista completa de todos os *endpoints* do Schema Registry API pode ser encontrada em https://docs.confluent.io/platform/current/schema-registry/develop/api.html

E o JSON Schema?

JSON Schema é uma especificação que tem como objetivo garantir que dados nesse formato atendam a um conjunto de requisitos predefinidos. Ela permite descrever de forma precisa a estrutura, o conteúdo e as restrições de um documento JSON, facilitando o controle de integridade dos dados, principalmente em sistemas de validação, comunicação entre APIs e armazenamento de dados.

Uma explicação detalhada está na página https://json-schema.org/docs, onde estão reunidas as especificações elaboradas ao longo dos anos. A última versão documentada até a redação deste livro é datada de dezembro de 2020.

Atualização de *schemas*

É possível que cada *subject* tenha diversos *schemas*. Veremos como atualizá-los no capítulo 7.

3.7 Resumo

Neste capítulo, vimos diversos *endpoints* de cadastro e busca da Schema Registry API, tomando como referência aqueles utilizados pelo Redpanda Console.

No próximo capítulo, voltaremos ao *ms-pedidos* e realizaremos as devidas correções no consumo de mensagens enviadas pelo *ms-restaurantes*.

CAPÍTULO 4

Implementando Schema Registry no consumidor de mensagens

Nos capítulos anteriores, exploramos diversas características do Schema Registry, como sua API e a criação de produtores de mensagens que o utilizam. Agora, precisamos corrigir o *ms-pedidos*, fazendo com que ele também utilize a ferramenta para validar as mensagens recebidas.

> Os códigos apresentados neste capítulo também podem ser encontrados na branch `cap-4`.

4.1 O ms-pedidos em detalhe

Assim como no *ms-restaurantes*, o *ms-pedidos* possui uma classe `KafkaObjects` que instancia todos os *beans* necessários para que as integrações com o Kafka e com o Schema Registry funcionem corretamente. A classe `KafkaObjects` completa encontra-se a seguir:

```
package br.com.alura.mspedidos.infra.kafka;
```

```
import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import
org.apache.kafka.clients.consumer.ConsumerConfig;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.kafka.config.ConcurrentKafkaListene
rContainerFactory;
import
org.springframework.kafka.config.KafkaListenerContainer
Factory;
import
org.springframework.kafka.core.DefaultKafkaConsumerFact
ory;
import
org.springframework.kafka.listener.ConcurrentMessageLis
tenerContainer;
import
org.springframework.kafka.support.serializer.JsonDeseri
alizer;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaObjects {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.consumer.group-id}")
    private String groupId;
```

```
    @Value("${spring.kafka.consumer.auto-offset-
reset}")
    private String autoOffsetReset;

    @Bean

KafkaListenerContainerFactory<ConcurrentMessageListener
Container<IdentificacaoItem, ItemCardapioRecebido>>
kafkaListener() {

ConcurrentKafkaListenerContainerFactory<IdentificacaoIt
em, ItemCardapioRecebido> factory =
                new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(
                consumerConfigs(),

jsonDeserializer(IdentificacaoItem.class),

jsonDeserializer(ItemCardapioRecebido.class)
        ));
        return factory;
    }

    private <T> JsonDeserializer<T>
jsonDeserializer(Class<T> classe){
        return new JsonDeserializer<T>(classe, false);
    }

    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
groupId);
```

```
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset);

props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI
G, JsonDeserializer.class);

        return props;
    }
}
```

Essa é uma maneira de organizar a criação de *beans*: todos os necessários para uma integração podem ser colocados em uma única classe, facilitando sua localização. Entretanto, essa organização não é obrigatória, e a criação dos objetos pode ser feita em diversas classes.

Também temos as classes que representam as mensagens:

- a `IdentificacaoItem` , que representa a identificação de uma mensagem;

- a `ItemCardapioRecebido`, que representa a mensagem propriamente dita.

Essas classes (e seus atributos e operações) estão representadas no diagrama:

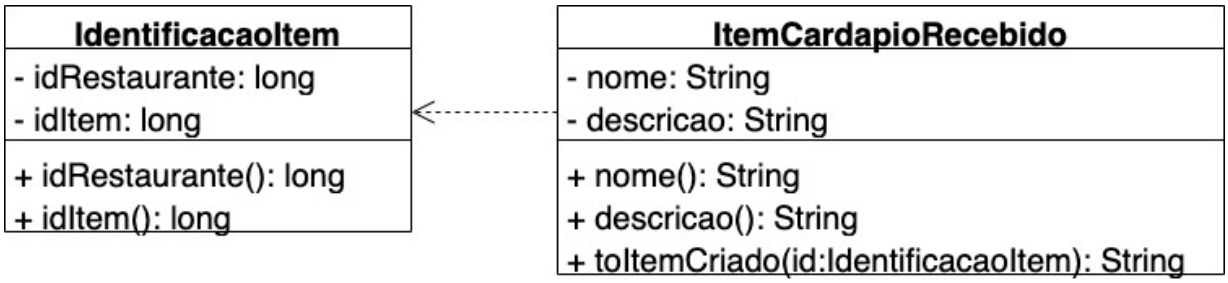


Figura 4.1: Relação entre classes que representam mensagens.

O código de `IdentificacaoItem` está a seguir:

```
package br.com.alura.mspedidos.dominio;

import com.fasterxml.jackson.annotation.JsonProperty;

public record IdentificacaoItem (@JsonProperty long
idRestaurante,
                                 @JsonProperty long
idItem) {
}
```

E o de `ItemCardapioRecebido` está a seguir:

```
package br.com.alura.mspedidos.infra.kafka;

import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import br.com.alura.mspedidos.dominio.ItemCardapio;
import
```

```
br.com.alura.mspedidos.dominio.ItemCardapioCriado;
import com.fasterxml.jackson.annotation.JsonProperty;

public record ItemCardapioRecebido(@JsonProperty String
nome,
                                   @JsonProperty String
descricao) {

    public ItemCardapioCriado
toItemCriado(IdentificacaoItem id) {
        return new ItemCardapioCriado(id, new
ItemCardapio(id.idItem(), nome, descricao));
    }
}
```

Repare que as classes que encapsulam a mensagem no *ms-pedidos* possuem uma estrutura muito semelhante às daquelas existentes no *ms-restaurantes*. E veremos que os ajustes que faremos aqui para podermos trabalhar com o Schema Registry também serão parecidos com os que fizemos no microsserviço anterior.

4.2 De JsonDeserializer para KafkaJsonSchemaDeserializer

No capítulo 2, alteramos os serializadores definidos no `KafkaObjects` de *ms-restaurantes*:

- de `JsonSerializer`

- para `KafkaJsonSchemaSerializer`

Dessa forma, conectamos o Schema Registry ao produtor, permitindo a validação das mensagens. De forma análoga no *ms-pedidos*, faremos alterações semelhantes no `KafkaObjects`:

- de `JsonDeserializer`
- para `KafkaJsonSchemaDeserializer`

Logo, tanto para a chave quanto para o valor, alteramos os desserializadores para ficarem assim:

```
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI
G, KafkaJsonSchemaDeserializer.class);
```

Também inserimos a dependência a seguir no *ms-restaurantes*, para que seja possível usar o desserializador mencionado. Portanto, é necessário adicioná-lo também no `pom.xml` do *ms-pedidos*:

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-json-schema-
serializer</artifactId>
    <version>${confluent.version}</version>
</dependency>
```

E inserir o valor da propriedade `${confluent.version}` juntamente às demais, também no `pom.xml`:

```
<properties>
    <java.version>21</java.version>
    <confluent.version>7.9.1</confluent.version>
</properties>
```

Precisamos também inserir a URL do Schema Registry que vamos usar para verificar as mensagens recebidas.

Para isso, usamos a propriedade `AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG`. E, assim como fizemos com a URL do Kafka, inserimos a URL do Schema Registry em um atributo de `KafkaObjects` que terá seu valor injetado pelo Spring.

O atributo é o `schemaRegistryUrl`, declarado como:

```
@Value("${spring.kafka.schema.registry.url}")
private String schemaRegistryUrl;
```

O atributo da anotação `@Value` precisa ser declarado no `application.yml`:

```
spring:
  application.name: ms-pedidos

  data.mongodb:
    uri:
mongodb://user:password@localhost:27017/restaurantes
    database: restaurantes

  kafka:
```

```
    bootstrap-servers: localhost:9092
    consumer:
      group-id: ms-pedidos
      auto-offset-reset: earliest
    schema.registry.url: http://localhost:8081
```

Para associar a propriedade com o seu respectivo valor, incluímos essa entrada no mapa de propriedades da classe `KafkaObjects`, como no exemplo a seguir:

```
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);
```

4.3 As constantes JSON_KEY_TYPE e JSON_VALUE_TYPE de KafkaJsonSchemaDeserializerConfig

Agora que configuramos o Schema Registry e alteramos o desserializador do consumidor, precisamos informar os tipos a serem usados como chave e valor na desserialização da mensagem.

Até o momento, para fazermos isso, usávamos o código a seguir presente em `KafkaObjects`:

```
@Bean
ConsumerFactory<IdentificacaoItem,
ItemCardapioRecebido> consumerFactory() {
    return new DefaultKafkaConsumerFactory<>(
            consumerConfigs(),
```

```
            jsonDeserializer(IdentificacaoItem.class),

jsonDeserializer(ItemCardapioRecebido.class));
}

private <T> JsonDeserializer<T>
jsonDeserializer(Class<T> classes){
    return new JsonDeserializer<T>(classes, false);
}

public Map<String, Object> consumerConfigs(Class<?>
...classes) {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset);

    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

    //as próximas 2 linhas foram alteradas neste
capítulo, para usar o novo desserializador

props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaDeserializer.class);

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI
G, KafkaJsonSchemaDeserializer.class);

    return props;
}
```

Repare que informávamos tanto o tipo da chave (`IdentificacaoItem`) quanto o tipo do valor (`ItemCardapioRecebido`) na construção do

desserializador a ser passado para o `DefaultKafkaConsumerFactory`. Dessa forma, os desserializadores eram construídos com as informações necessárias para desserializar tanto a chave quanto o valor da mensagem sem incidentes.

Podemos, entretanto, informar os tipos diretamente no mapa de propriedades. Para o `KafkaJsonSchemaDeserializer` usá-los, devemos considerar as seguintes constantes:

- `KafkaJsonSchemaDeserializerConfig.JSON_KEY_TYPE`: para informar o tipo da chave da mensagem
- `KafkaJsonSchemaDeserializerConfig.JSON_VALUE_TYPE`: para informar o tipo do valor da mensagem

Os valores a serem usados em ambas as constantes são os nomes das classes. Assim, o mapa de propriedades deve ser atualizado da seguinte forma:

```
props.put(KafkaJsonSchemaDeserializerConfig.JSON_KEY_TY
PE, IdentificacaoItem.class.getCanonicalName());
props.put(KafkaJsonSchemaDeserializerConfig.JSON_VALUE_
TYPE, ItemCardapioRecebido.class.getCanonicalName());
```

Com essas configurações, podemos remover o método `jsonDeserializer()`, fazendo com que a classe `KafkaObjects` fique como mostrado a seguir:

```
package br.com.alura.mspedidos.infra.kafka;
```

```
import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import
io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe
Config;
import
io.confluent.kafka.serializers.json.KafkaJsonSchemaDese
rializer;
import
io.confluent.kafka.serializers.json.KafkaJsonSchemaDese
rializerConfig;
import
org.apache.kafka.clients.consumer.ConsumerConfig;
import
org.apache.kafka.common.serialization.Deserializer;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.kafka.config.ConcurrentKafkaListene
rContainerFactory;
import
org.springframework.kafka.config.KafkaListenerContainer
Factory;
import
org.springframework.kafka.core.DefaultKafkaConsumerFact
ory;
import
org.springframework.kafka.listener.ConcurrentMessageLis
tenerContainer;

import java.util.HashMap;
import java.util.Map;
```

```
@Configuration
public class KafkaObjects {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.consumer.group-id}")
    private String groupId;

    @Value("${spring.kafka.consumer.auto-offset-
reset}")
    private String autoOffsetReset;

    @Value("${spring.kafka.schema.registry.url}")
    private String schemaRegistryUrl;

    @Bean

KafkaListenerContainerFactory<ConcurrentMessageListener
Container<IdentificacaoItem, ItemCardapioRecebido>>
kafkaListener() {

ConcurrentKafkaListenerContainerFactory<IdentificacaoIt
em, ItemCardapioRecebido> factory =
                new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(
                consumerConfigs()
        ));
        return factory;
    }

    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
```

```
groupId);

props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset);

props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaDeserializer.class);

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI
G, KafkaJsonSchemaDeserializer.class);

props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(KafkaJsonSchemaDeserializerConfig.JSON_KEY_TY
PE, IdentificacaoItem.class.getCanonicalName());

props.put(KafkaJsonSchemaDeserializerConfig.JSON_VALUE_
TYPE, ItemCardapioRecebido.class.getCanonicalName());

        return props;
    }
}
```

4.4 Corrigindo os testes

Há também o `KafkaObjectsParaTestes`, classe semelhante à anterior, mas que cria objetos Kafka apenas para fins de teste.

No caso do *ms-pedidos*, precisamos apenas de um objeto que envie mensagens para o tópico Kafka, para que possamos testar o consumidor de mensagens. Usaremos o `KafkaTemplate`, já usado no *ms-restaurantes*. Sua montagem está no código a seguir:

```
package br.com.alura.mspedidos.infra.kafka;

import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import br.com.alura.mspedidos.dominio.ItemCardapio;
import
org.apache.kafka.clients.producer.ProducerConfig;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.kafka.core.DefaultKafkaProducerFact
ory;
import org.springframework.kafka.core.KafkaTemplate;
import
org.springframework.kafka.support.serializer.JsonSerial
izer;

import java.util.HashMap;
import java.util.Map;

@Configuration
class KafkaObjectsParaTestes {

    @Value("${spring.kafka.bootstrap-servers}")
```

```
    private String kafkaServerUrl;

    @Bean
    KafkaTemplate<IdentificacaoItem,
ItemCardapioRecebido> kafkaTemplate(){
        return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(produtorConfigs()));
    }

    private Map<String, Object> produtorConfigs() {
        Map<String, Object> props = new HashMap<>();

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        return props;
    }
}
```

Precisamos ajustar o objeto `KafkaTemplate` para passar a trabalhar com Schema Registry. Para isso:

- de forma semelhante ao que fizemos no código de produção, substituímos `JsonSerializer` por `KafkaJsonSchemaSerializer` no mapa de propriedades:

```
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);
```

```
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);
```

- inserimos o `KafkaJsonSchemaSerializer` como serializador de mensagens tanto na chave da mensagem quanto na mensagem em si:

```
@Bean
KafkaTemplate<IdentificacaoItem, ItemCardapioRecebido>
kafkaTemplate(){
    return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(
            produtorConfigs()
            , new KafkaJsonSchemaSerializer<>()
            , new KafkaJsonSchemaSerializer<>()
    ));
}
```

O primeiro parâmetro do construtor usado aqui de `DefaultKafkaProduceFactory` é o mapa de propriedades. Entretanto, inserimos mais dois parâmetros: o segundo representa o serializador a ser usado na chave da mensagem e o terceiro, no valor da mensagem. Ambos são instâncias de `KafkaJsonSchemaSerializer`.

Ao rodarmos os testes da aplicação, percebemos um erro semelhante ao encontrado no capítulo 2, no teste existente na classe `ReceptorTest`:

```
org.apache.kafka.common.errors.SerializationException:
Error serializing JSON message

    at
io.confluent.kafka.serializers.json.AbstractKafkaJsonSc
hemaSerializer.serializeImpl(AbstractKafkaJsonSchemaSer
ializer.java:173)
    at
io.confluent.kafka.serializers.json.KafkaJsonSchemaSeri
alizer.serialize(KafkaJsonSchemaSerializer.java:95)
    at
org.apache.kafka.clients.producer.KafkaProducer.doSend(
KafkaProducer.java:1017)
    at
org.apache.kafka.clients.producer.KafkaProducer.send(Ka
fkaProducer.java:964)
    at
org.springframework.kafka.core.DefaultKafkaProducerFact
ory$CloseSafeProducer.send(DefaultKafkaProducerFactory.
java:1103)
    at
org.springframework.kafka.core.KafkaTemplate.doSend(Kaf
kaTemplate.java:825)
    at
org.springframework.kafka.core.KafkaTemplate.observeSen
d(KafkaTemplate.java:793)
    at
org.springframework.kafka.core.KafkaTemplate.send(Kafka
Template.java:576)
    at
br.com.alura.mspedidos.infra.kafka.ReceptorTest.receber
UmNovoItemCardapio(ReceptorTest.java:50)
    at
java.base/java.lang.reflect.Method.invoke(Method.java:5
80)
    at
```

```
java.base/java.util.ArrayList.forEach(ArrayList.java:15
96)
    at
java.base/java.util.ArrayList.forEach(ArrayList.java:15
96)
Caused by: java.io.IOException: Incompatible schema
'{"$schema":"http://json-schema.org/draft/2020-
12/schema#","title":"Item
Cardapio","type":"object","additionalProperties":false,
"properties":{"nome":{"type":"string"},"descricao":
{"type":"string"}},"required":["nome","descricao"]}'
with refs '[]' of type 'JSON' for schema
'{"$schema":"http://json-schema.org/draft-
07/schema#","title":"Item Cardapio
Recebido","type":"object","additionalProperties":false,
"properties":{"nome":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"string"}]},"descricao":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"string"}]}}}'. Set
latest.compatibility.strict=false to disable this
check. See log file for more details.
    at
io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe
.lookupLatestVersion(AbstractKafkaSchemaSerDe.java:625)
    at
io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe
.lookupLatestVersion(AbstractKafkaSchemaSerDe.java:586)

// trechos omitidos por brevidade
```

As causas são as mesmas encontradas no capítulo 2: para o Schema Registry, é necessário informar que os campos da mensagem são obrigatórios, pois é o que está escrito no *schema*.

Vamos então inserir o atributo `required` com o valor `true` nos campos da mensagem:

```
package br.com.alura.mspedidos.infra.kafka;

import
br.com.alura.mspedidos.dominio.IdentificacaoItem;
import br.com.alura.mspedidos.dominio.ItemCardapio;
import
br.com.alura.mspedidos.dominio.ItemCardapioCriado;
import com.fasterxml.jackson.annotation.JsonProperty;

public record
ItemCardapioRecebido(@JsonProperty(required = true)
String nome,

@JsonProperty(required = true) String descricao) {

    public ItemCardapioCriado
toItemCriado(IdentificacaoItem id) {
        return new ItemCardapioCriado(id, new
ItemCardapio(id.idItem(), nome, descricao));
    }
}
```

Se rodarmos os testes novamente, eles passam:

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped:
0
[INFO]
[INFO] ------------------------------------------------
------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
------------------------
[INFO] Total time:  41.323 s
[INFO] Finished at: 2025-02-02T18:18:36-03:00
[INFO] ------------------------------------------------
------------------------

Process finished with exit code 0
```

> É possível que as refatorações explicadas neste capítulo (e nos demais) sejam feitas montando os *beans* de maneiras diferentes, e a aplicação continue funcionando. Para que esse processo seja possível, recorremos à escrita de testes automatizados, não somente nesse microsserviço, mas em todos os demais, comprovando o seu correto funcionamento.

Entretanto, alteramos apenas a classe que representa a mensagem, e não a que representa a chave da mensagem. E mesmo assim os testes passam! Veremos a seguir por que isso ocorre.

4.5 Tipos primitivos nos atributos das mensagens e seus problemas

Assim como ocorreu no capítulo 2, temos a manipulação de tipos primitivos nas mensagens, gerando um comportamento anômalo. Não alteramos a classe `IdentificacaoItem` e mesmo assim os testes passam.

Essa falha ocorre devido à incompatibilidade entre o Java e o JSON. Enquanto o primeiro lida com tipos primitivos, o segundo não, podendo haver conflitos de conversão dos atributos das mensagens. Por outro lado, o Java tem dificuldades para converter `null` (que o JSON suporta) em um tipo primitivo, por exemplo.

Isso ocorre devido a conflitos de incompatibilidade entre o Java e o JSON:

- O JSON permite valores `null`, mas os tipos primitivos em Java não aceitam `null`;
- Se um campo opcional estiver ausente na mensagem JSON e o atributo Java for primitivo, pode haver problemas na desserialização.

Vamos ver um exemplo com uma possível mensagem que será desserializada para um objeto da classe `IdentificacaoItem`:

```
{
  "idRestaurante": 1,
```

```
  "idItem": null
}
```

Se o `idItem` for do tipo `int`, isso causará um erro; mas se for do tipo `Integer`, funcionará normalmente.

Para esses problemas, é importante que as mensagens sempre usem os tipos *wrapper* no lugar de primitivos (como `Integer`, `Long`, `Double` entre outros). Vamos corrigir o `IdentificacaoItem`, substituindo os atributos `long` por `Long`. O código fica assim:

```
package br.com.alura.mspedidos.dominio;

import com.fasterxml.jackson.annotation.JsonProperty;

public record IdentificacaoItem (@JsonProperty Long
idRestaurante,
                                 @JsonProperty Long
idItem) {
}
```

Ao rodarmos os testes, o incidente de incompatibilidade de *schemas* volta a aparecer, o que faz mais sentido:

```
Caused by: java.io.IOException: Incompatible schema
'{"$schema":"http://json-schema.org/draft/2020-
12/schema#","title":"Identificacao
Item","type":"object","additionalProperties":false,"pro
perties":{"idRestaurante":{"type":"number"},"idItem":
{"type":"number"}},"required":
```

```
["idRestaurante","idItem"]}' with refs '[]' of type
'JSON' for schema '{"$schema":"http://json-
schema.org/draft-07/schema#","title":"Identificacao
Item","type":"object","additionalProperties":false,"pro
perties":{"idRestaurante":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"integer"}]},"idItem":{"oneOf":
[{"type":"null","title":"Not included"},
{"type":"integer"}]}}}'. Set
latest.compatibility.strict=false to disable this
check. See log file for more details.
    at
io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe
.lookupLatestVersion(AbstractKafkaSchemaSerDe.java:625)
    at
io.confluent.kafka.serializers.AbstractKafkaSchemaSerDe
.lookupLatestVersion(AbstractKafkaSchemaSerDe.java:586)
    at
io.confluent.kafka.serializers.json.AbstractKafkaJsonSc
hemaSerializer.serializeImpl(AbstractKafkaJsonSchemaSer
ializer.java:153)
    ... 11 more
```

A correção é a mesma realizada na classe ItemCardapioRecebido :

```
package br.com.alura.mspedidos.dominio;

import com.fasterxml.jackson.annotation.JsonProperty;

public record IdentificacaoItem (@JsonProperty(required
= true) Long idRestaurante,
                                 @JsonProperty(required
= true) Long idItem) {
}
```

Ao rodarmos os testes, todos continuam passando sem problemas. E agora, com os tipos recomendados para os atributos.

4.6 Resumo

Vimos neste capítulo como criar consumidores de mensagens usando JSON Schema. No próximo capítulo, veremos como trabalhar com outros formatos de dados, além do JSON, para envio e recebimento de mensagens.

Capítulo 5

O Avro como formato de mensagens

Até o momento, usamos o JSON como formato para as mensagens. Por ser baseado em texto, é de fácil compreensão por humanos. Entretanto, possui alguns problemas de desempenho quando há necessidade de processar grandes volumes de mensagens por hora ou ao longo do dia. Felizmente, o Schema Registry também oferece suporte a outros formatos de mensagem — binários e, portanto, menores e mais fáceis de serem processados por computadores — como o Avro. A partir deste capítulo, veremos como enviar e receber mensagens utilizando esse formato.

5.1 Os problemas do formato JSON, e o Avro como solução

Para alguns tipos de sistemas, em que a *performance* não é um requisito tão preocupante, ou mesmo o volume de processamento de mensagens não seja tão alto, os protocolos baseados em texto podem ser usados sem prejuízos. No entanto, se o volume de mensagens crescer significativamente,

formatos como JSON podem apresentar limitações, como as descritas a seguir:

- Tamanho da mensagem: formatos baseados em texto são mais verbosos se comparados a formatos binários. Além disso, todos os nomes de campos estão presentes na mensagem.
- Validação mais complexa: apesar de o Schema Registry oferecer suporte ao JSON, a validação ocorre apenas na estrutura, sem garantir uma coerência perfeita de tipos. Isso pode resultar em falhas silenciosas, como observado em capítulos anteriores, quando surgiram problemas quando trabalhamos com tipos primitivos.
- Custo computacional: como JSON é um formato textual, ele exige mais processamento para serialização e desserialização, se comparado com protocolos binários.

Para entender melhor os problemas relacionados a formatos baseados em texto, recomendo a audição da edição 17 do *podcast*, disponível em https://www.hipsters.tech/microservicos-hipsters-17/. Neste episódio, Phillip Calçado (também conhecido como "Phil" Calçado) comenta que uma das empresas que ele trabalhou (que possuía cerca de 350 milhões de usuários por mês) após a realização de algumas análises, verificou que entre 10% a 20% do tempo de CPU de todas as máquinas e aplicações era gasto com leitura e escrita de JSON, fazendo com que a empresa gastasse milhares de dólares por mês apenas com *parsing* de mensagens.

O Apache Avro é um sistema de serialização de dados originalmente desenvolvido como parte do projeto Apache Hadoop. Foi criado para fornecer uma solução eficiente e flexível para o armazenamento e transmissão de dados estruturados. É amplamente utilizado em sistemas distribuídos, incluindo o Apache Kafka. Ele foi projetado para ser altamente compacto, com suporte a *schemas* dinâmicos e de fácil integração com diversas linguagens de programação.

Uma das principais características do Avro é seu esquema autodescritivo, que permite que os dados sejam lidos e interpretados corretamente mesmo quando o *schema* evolui. Esse recurso, combinado com a compatibilidade avançada oferecida pelo Schema Registry, torna o Avro uma escolha ideal para sistemas que precisam lidar com grandes volumes de dados e evolução de mensagens sem comprometer a compatibilidade.

Consumidores de tópicos Kafka que possuem mensagens no formato JSON necessitam do apoio do Schema Registry para que seja possível validar as mensagens, visto que o JSON Schema não é embutido na mensagem (considerando que as mensagens precisam de validação). Já no caso do Avro, temos duas alternativas:

- embutir o *schema* na mensagem: isso permite sua evolução sem causar problemas nos consumidores, desde que eles estejam preparados para essa característica;
- referenciar o *schema* na mensagem: nesse caso, o Schema Registry vai conter o *schema* da mensagem, e ela possuirá apenas uma referência a ele. Assim, o Schema Registry pode realizar a validação de cada mensagem recebida.

Para mais comparações relacionadas a esses formatos, recomenda-se a leitura dos *links*:

- https://www.confluent.io/blog/avro-kafka-data/
- https://docs.confluent.io/platform/current/schema-registry/fundamentals/serdes-develop/index.html
- https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

5.2 Realização de pedidos no ms-pedidos

O código referente a este capítulo encontra-se na *branch* `cap-5`.

Já implementamos a criação de cardápio para os restaurantes (via `ms-restaurantes`) e o envio desses para o `ms-pedidos`. Com esses requisitos implementados, podemos pensar na implementação de funcionalidades que permitam aos clientes realizar pedidos de compra de comida.

Para isso, implementamos dois *controllers* bem simples. Um deles exibe os cardápios já criados,

desde que fornecida a identificação do restaurante:

```
package br.com.alura.mspedidos.controller;

import br.com.alura.mspedidos.dominio.Cardapio;
import br.com.alura.mspedidos.service.CardapioService;
import lombok.AllArgsConstructor;
import org.springframework.http.ResponseEntity;
import
org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/cardapios")
@AllArgsConstructor
class CardapioController {

    private CardapioService cardapioService;

    @GetMapping("/{idRestaurante}")
    public ResponseEntity<Cardapio>
exibirCardapio(@PathVariable("idRestaurante") Long
idRestaurante){
        return
ResponseEntity.of(cardapioService.obterDoRestaurante(id
Restaurante));
    }
}
```

O método `obterDoRestaurante()` presente em `CardapioService` simplesmente repassa a tarefa para um objeto da interface `TodosCardapios`:

```
public Optional<Cardapio> obterDoRestaurante(Long
idRestaurante) {
    return
todosCardapios.findByIdRestaurante(idRestaurante);
}
```

O outro *controller* é utilizado para a criação do pedido em si, o `PedidosController`. Ele recebe o pedido e:

- o persiste em banco de dados;
- o envia para um tópico chamado `pedidos`. Esse tópico será lido por um microsserviço a ser implementado: o *ms-entregadores*, que receberá o pedido e selecionará o entregador que transportará o pedido.

> O *ms-entregadores*, bem o consumo de mensagens a ser realizado por ele, serão implementados no capítulo 6.

Para criar o tópico `pedidos`, basta inserir a linha abaixo no `docker-compose.yml`, juntamente da linha que cria o tópico `restaurantes.cardapio.itens`:

```
kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic pedidos
```

A seguir, mostramos como será feita a criação de pedidos e como ocorrerá a integração entre o *ms-pedidos* e o *ms-entregadores*:

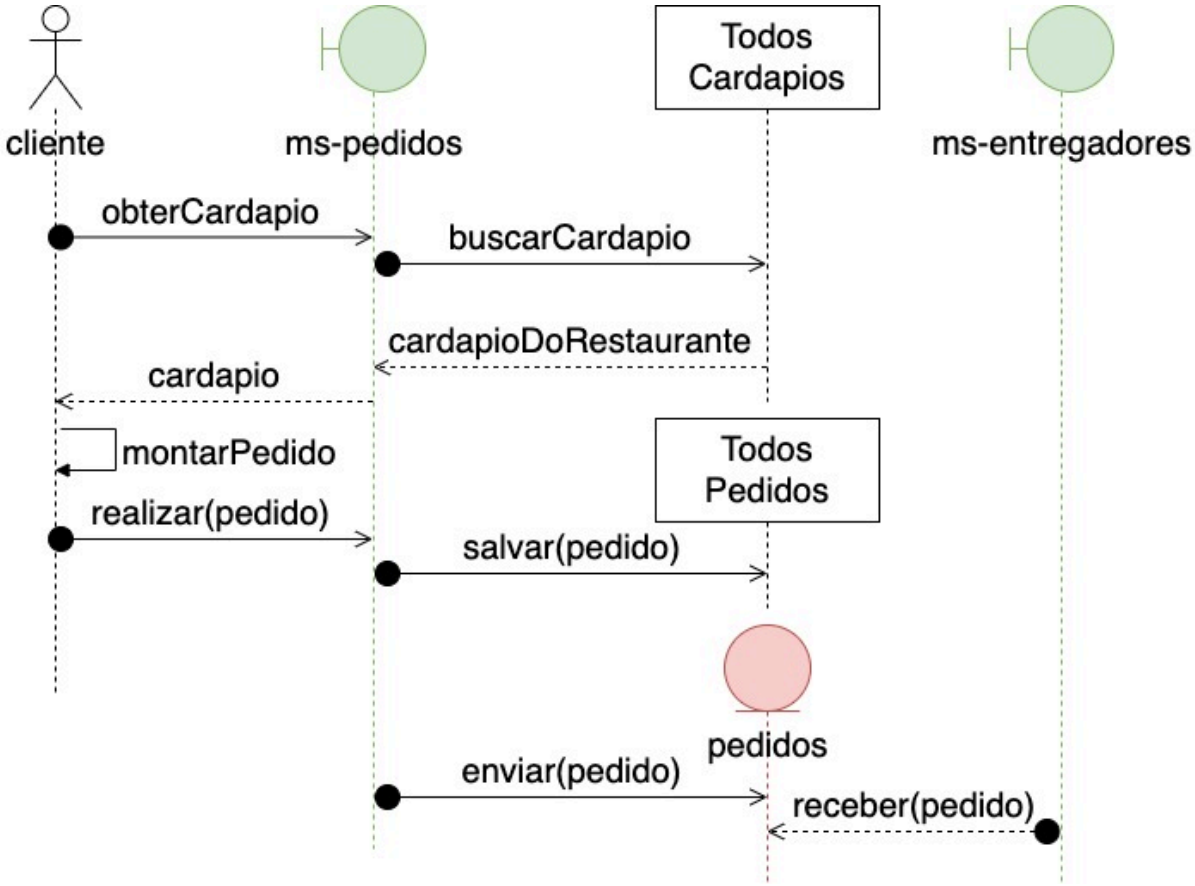


Figura 5.1: Integração entre o ms-pedidos e o ms-entregadores.

Assim como o `CardapioController`, a estrutura do `PedidosController` é simples, contendo apenas um `PedidoService`, que por sua vez possui duas dependências:

- a interface `TodosPedidos`, responsável por armazenar o pedido no banco de dados;
- a interface `PedidosSender`, que contém o contrato a ser implementado pela classe que fará o envio da mensagem.

O `PedidosController` está implementado conforme o código a seguir:

```
package br.com.alura.mspedidos.controller;

import br.com.alura.mspedidos.dominio.Pedido;
import br.com.alura.mspedidos.service.PedidoService;
import lombok.AllArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.web.bind.annotation.PostMapping;
import
org.springframework.web.bind.annotation.RequestBody;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/pedidos")
@AllArgsConstructor
class PedidosController {

    private PedidoService pedidoService;

    @PostMapping
    public ResponseEntity<Pedido>
```

```
criarPedido(@RequestBody PedidoCriado pedidoCriado) {

pedidoService.criarPedido(pedidoCriado.pedido());

        return
ResponseEntity.status(HttpStatus.CREATED).build();
    }
}
```

O `PedidoService` possui o seguinte código:

```
package br.com.alura.mspedidos.service;

import br.com.alura.mspedidos.dominio.Pedido;
import br.com.alura.mspedidos.dominio.PedidosSender;
import br.com.alura.mspedidos.dominio.TodosPedidos;
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@AllArgsConstructor
public class PedidoService {

    private PedidosSender pedidosSender;
    private TodosPedidos todosPedidos;

    public void criarPedido(Pedido pedido) {
        todosPedidos.save(pedido);
        pedidosSender.enviar(pedido);
    }
}
```

A interface `PedidoSender` é implementada pela classe `PedidosSenderKafka`, que veremos a seguir.

5.3 Implementação do formato Avro na comunicação entre os microsserviços usando AVSC (Avro Schema)

Para construir mensagens no formato Avro, é necessário primeiro definir seu *schema*. Há duas maneiras de se fazer isso:

- usando o formato de definição de *schema* AVSC. Ele baseia-se na linguagem JSON para definir os campos de uma mensagem, bem como seus tipos e restrições;
- usando o formato IDL (*Interface Definition Language*). Usa uma linguagem declarativa exclusiva de criação de *schemas*.

> Neste capítulo usaremos o formato AVSC.
> Veremos o IDL no capítulo 6.

Para facilitar a organização dos pedidos a serem enviados pelo *ms-pedidos* para o tópico `pedidos`, usaremos a estrutura de objetos a seguir:

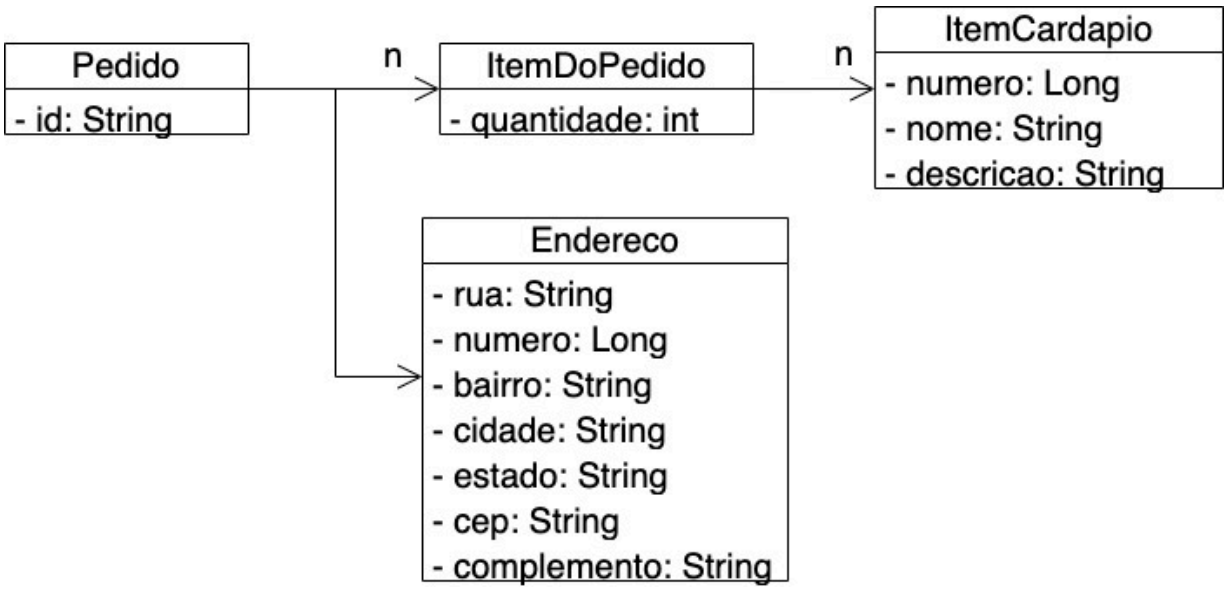


Figura 5.2: Objetos participantes da criação de uma mensagem que represente um pedido.

Assim como há relação entre objetos Java e JSON, também existe essa correspondência entre arquivos AVSC e as classes Java a serem usadas na produção de uma mensagem. Podemos usar um arquivo AVSC por projeto. Assim, podemos evoluir esses arquivos separadamente.

Os tipos primitivos do Avro

O protocolo Avro possui os seguintes tipos primitivos:

- `null` : representa nenhum valor;
- `boolean` : um valor binário;
- `int` : um inteiro de 32 bits;

- `long` : um inteiro de 64 bits;
- `float` : número de ponto flutuante de precisão única (32 bits);
- `double` : número de ponto flutuante de dupla precisão (64 bits);
- `bytes` : sequência de bytes não assinados de 8 bits;
- `string` : sequência de caracteres unicode.

É possível fazer uma comparação direta com os tipos primitivos do Java, como `int`, `float` ou `string` (que são equivalentes aos tipos `int`, `float` e `String`).

Vamos então escrever os *schemas* dos tipos "folha" do diagrama mostrado: os tipos `Endereco` e `ItemCardapio`. A classe `Endereco` possui diversos atributos:

```
public class Endereco {
    private String rua;
    private Integer numero;
    private String bairro;
    private String cidade;
    private String estado;
    private String cep;
    private String complemento;
```

Criamos um arquivo `Endereco.avsc` na pasta `src/main/avro`, com os atributos descritos representados:

```
{
  "name": "rua",
  "type": "string"
},
{
  "name": "numero",
  "type": "int"
},
{
  "name": "bairro",
  "type": "string"
},
{
  "name": "cidade",
  "type": "string"
},
{
  "name": "estado",
  "type": "string"
},
{
  "name": "cep",
  "type": "string"
}
```

O último campo a ser representado é o de complemento. Em um endereço, essa informação não costuma ser obrigatória, visto que nem todo endereço o contém. Então, vamos declará-lo como opcional no *schema*. Para isso, indicamos no atributo *type* que o campo pode aceitar dois tipos: o `string` e o `null`.

A declaração desse campo ficará assim, portanto:

```
{
  "name": "complemento",
  "type": ["null", "string"],
  "default": null
}
```

O atributo `default` informa o valor padrão para esse campo. No caso, `null`.

O tipo record

Para declarar a classe `Endereco` (ou diversas outras classes em Java), precisamos não somente declarar seus atributos, mas também indicar que esses atributos pertencem a uma classe com nome e pacotes definidos. Com o Avro, isso não é diferente.

Em Java, declaramos pacote e classe conforme o exemplo a seguir:

```
package br.com.alura.mspedidos.dominio;

// trechos de código omitidos

public class Endereco {
```

No formato AVSC, utilizamos o tipo complexo `record`, que contém os seguintes atributos:

- `namespace` : define um "espaço de nomes" para organizar os *schemas* e evitar conflitos de nomes. Semelhante ao `package` do Java.
- `name` : define o nome do tipo de dado. Observe que há uma correspondência com o nome da classe em Java.
- `type` : por se tratar de um `record` , possui valor fixo `record` . Entretanto, já vimos esse atributo também sendo usado para informar os tipos dos atributos de `Endereco` . Nesses casos, naturalmente ele deve variar.
- `fields` : define os atributos de um `record` .
- `doc` : pode ser usado para documentar o *schema*. É opcional.

Para introduzir esse tipo no *schema* de `Endereco` , vamos inserir o atributo `fields` , que agrupa os atributos já existentes:

```
"fields": [
  {
    "name": "rua",
    "type": "string"
  },
  {
    "name": "numero",
    "type": "int"
  },
  {
    "name": "bairro",
    "type": "string"
```

```
      },
      {
        "name": "cidade",
        "type": "string"
      },
      {
        "name": "estado",
        "type": "string"
      },
      {
        "name": "cep",
        "type": "string"
      },
      {
        "name": "complemento",
        "type": ["null", "string"],
        "default": null
      }
    ]
```

Por último, incluímos os demais atributos de `record`.

O `schema` completo está a seguir:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "doc": "O endereço onde o pedido deve ser entregue",
  "name": "Endereco",
    "fields": [
      {
        "name": "rua",
        "type": "string"
      },
      {
```

```
      "name": "numero",
      "type": "int"
    },
    {
      "name": "bairro",
      "type": "string"
    },
    {
      "name": "cidade",
      "type": "string"
    },
    {
      "name": "estado",
      "type": "string"
    },
    {
      "name": "cep",
      "type": "string"
    },
    {
      "name": "complemento",
      "type": ["null", "string"],
      "default": null
    }
  ]
}
```

Foram inseridas diversas quebras de linha e recuos para facilitar a leitura. Entretanto, essa formatação não é obrigatória.

O próximo *schema* é o `ItemCardapio.avsc`. Ele não é muito diferente do `Endereco.avsc`. Só possui menos

campos, além do uso do tipo `long`:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "doc": "Descreve um item do cardápio. Usado para
descrever detalhadamente um item de pedido",
  "name": "ItemCardapio",
    "fields": [
        {
        "name": "numero",
        "type": "long"
        },
        {
        "name": "nome",
        "type": "string"
        },
        {
        "name": "descricao",
        "type": "string"
        }
    ]
}
```

Referências a tipos nomeados

A classe `ItemDoPedido` possui dois atributos: um do tipo `int` e outro do tipo `ItemCardapio`:

```
public class ItemDoPedido {

    private ItemCardapio item;
```

```
    private int quantidade;
// trechos de código omitidos
```

Já sabemos que podemos representar a quantidade com um tipo `int` do Avro no *schema*:

```
{
  "name": "quantidade",
  "type": "int"
}
```

Para representar o atributo `itemCardapio`, temos duas alternativas:

- Copiar o conteúdo de `ItemCardapio.avsc` para `ItemDoPedido.avsc`. Isso exigiria o aninhamento da estrutura de `ItemCardapio.avsc` dentro de `ItemDoPedido.avsc`. O *schema* ficaria assim:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "name": "ItemDoPedido",
  "doc": "Representa um item do pedido, incluindo suas
informações do cardápio e a quantidade",
  "fields": [
    {
      "name": "itemCardapio",
      "type": {
        "type": "record",
        "name": "ItemCardapio",
```

```
        "doc": "Descreve um item do cardápio",
        "fields": [
          { "name": "numero", "type": "long" },
          { "name": "nome", "type": "string" },
          { "name": "descricao", "type": "string" }
        ]
      }
    },
    {
      "name": "quantidade",
      "type": "int",
      "doc": "Quantidade do item no pedido"
    }
  ]
}
```

- Fazer uma *referência a um tipo nomeado*, informando no atributo `type` o tipo existente em outro *schema*:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "name": "ItemDoPedido",
  "doc": "Representa um dos muitos produtos que um
pedido pode ter. Descreve também a quantidade de
produtos do mesmo tipo",
  "fields": [
    {
      "name": "itemCardapio",
      "type":
"br.com.alura.mspedidos.infra.kafka.avro.ItemCardapio"
    },
    {
```

```
      "name": "quantidade",
      "type": "int"
    }
  ]}
}
```

A primeira alternativa traz algumas desvantagens:

- Redundância: se o `ItemCardapio` for usado em outros lugares, a informação pode se repetir.
- Dificulta mudanças: qualquer alteração na estrutura de `ItemCardapio` precisa ser propagada manualmente.

Por isso, adotaremos a segunda alternativa.

O tipo complexo array

Por último, a classe `Pedido` possui os seguintes atributos, que também precisam ser mapeados para o *schema* Avro:

```
public class Pedido {

// trechos de código omitidos

    private Set<ItemDoPedido> itensDoPedido;
    private Endereco endereco;

// trechos de código omitidos
```

Repare que temos uma `Collection` do Java como tipo de um dos atributos da classe. No Avro, ela é

representada como um `array`. Mas mais do que isso: um `array` de `ItemDoPedido`. Sua declaração deve ser feita como a seguir:

```
{
  "name": "itensDoPedido",
  "type": {
    "type": "array",
    "items": {
      "type":
"br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedido"
    }
  }
}
```

O *schema* completo de `Pedido` está a seguir:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "name": "Pedido",
  "doc": "Representa uma compra de produtos de um mesmo
restaurante",
  "fields": [
    {
      "name": "itensDoPedido",
      "type": {
        "type": "array",
        "items": {
          "type":
"br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedido"
        }
      }
    },
```

```
    {
      "name": "endereco",
      "type":
"br.com.alura.mspedidos.infra.kafka.avro.Endereco"
    }
  ]
}
```

Existem outros tipos complexos além de `record` e `array`. Vale a pena consultar a documentação presente em https://avro.apache.org/docs/1.12.0/specification/ para essas e outras informações.

5.4 O plugin avro-maven-plugin

Para que possamos enviar mensagens para o Kafka no formato Avro, precisamos converter nossos objetos Java para esse formato. Já fizemos isso quando enviamos mensagens no formato JSON. Com JSON, bastava inserir algumas anotações nos atributos das classes, e a biblioteca Jackson cuidava da conversão.

No caso do Avro, podemos utilizar bibliotecas que criam classes Java a partir dos *schemas* que definimos. Isso é possível com o `avro-maven-plugin`,

um *plugin* do Maven. Essas classes são chamadas de *stubs*.

> Um *stub* é um objeto substituto que vai retornar respostas predefinidas para as chamadas de seus métodos. No caso do `avro-maven-plugin`, as classes criadas por ele não contêm lógica de negócio, mas apenas representam a estrutura do dado e possuem um contrato definido.

Basicamente, precisamos informar quais *schemas* devem participar do processo. O *plugin* vai gerar as classes diretamente na pasta `target` do projeto. Isso permitirá a sua utilização pelas demais classes. Vamos então:

- declarar o *plugin* no `pom.xml` do *ms-pedidos*, bem como a *phase* que será usada para a geração das classes (no caso, usarmos a `generate-sources`. Ou seja, as classes serão geradas ao rodar, no mínimo, o comando `mvn compile`).
- declarar os *schemas* a serem usados.

Também informamos que as classes geradas pelo *plugin* não devem conter *setters*. Veremos que as classes geradas possuem outros recursos para a criação de objetos dessas classes, tornando-os desnecessários.

Sobre os problemas relacionados à criação de *getters* e *setters*, recomendo a leitura do artigo disponível em https://www.alura.com.br/artigos/nao-aprender-oo-getters-e-setters

A declaração completa do plugin no `pom.xml` do *ms-pedidos* está a seguir:

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>${avro.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      <configuration>
          <createSetters>false</createSetters>
        <imports>

<import>${project.basedir}/src/main/resources/avro/Endereco.avsc</import>

<import>${project.basedir}/src/main/resources/avro/ItemCardapio.avsc</import>

<import>${project.basedir}/src/main/resources/avro/ItemDoPedido.avsc</import>

<import>${project.basedir}/src/main/resources/avro/Pedido.avsc</import>
        </imports>
```

```
      </configuration>
    </execution>
  </executions>
</plugin>
```

> **IMPORTANTE:** os *scripts* precisam estar na ordem em que devem ser processados.

Considere o *script* `Pedido.avsc`. Ele depende de `ItemDoPedido.avsc` (que por sua vez depende de `ItemCardapio.avsc`) e de `Endereco.avsc`. Assim que esses *scripts* forem processados, eles gerarão classes Java. Se isso não ocorrer antecipadamente, as dependências de `Pedido.avsc` não existirão, e consequentemente, a classe `Pedido` não poderá ser gerada.

Ao rodarmos o comando abaixo em *ms-pedidos*:

```
mvn -f ms-pedidos/pom.xml compile
```

Podemos notar que foram geradas as seguintes classes:

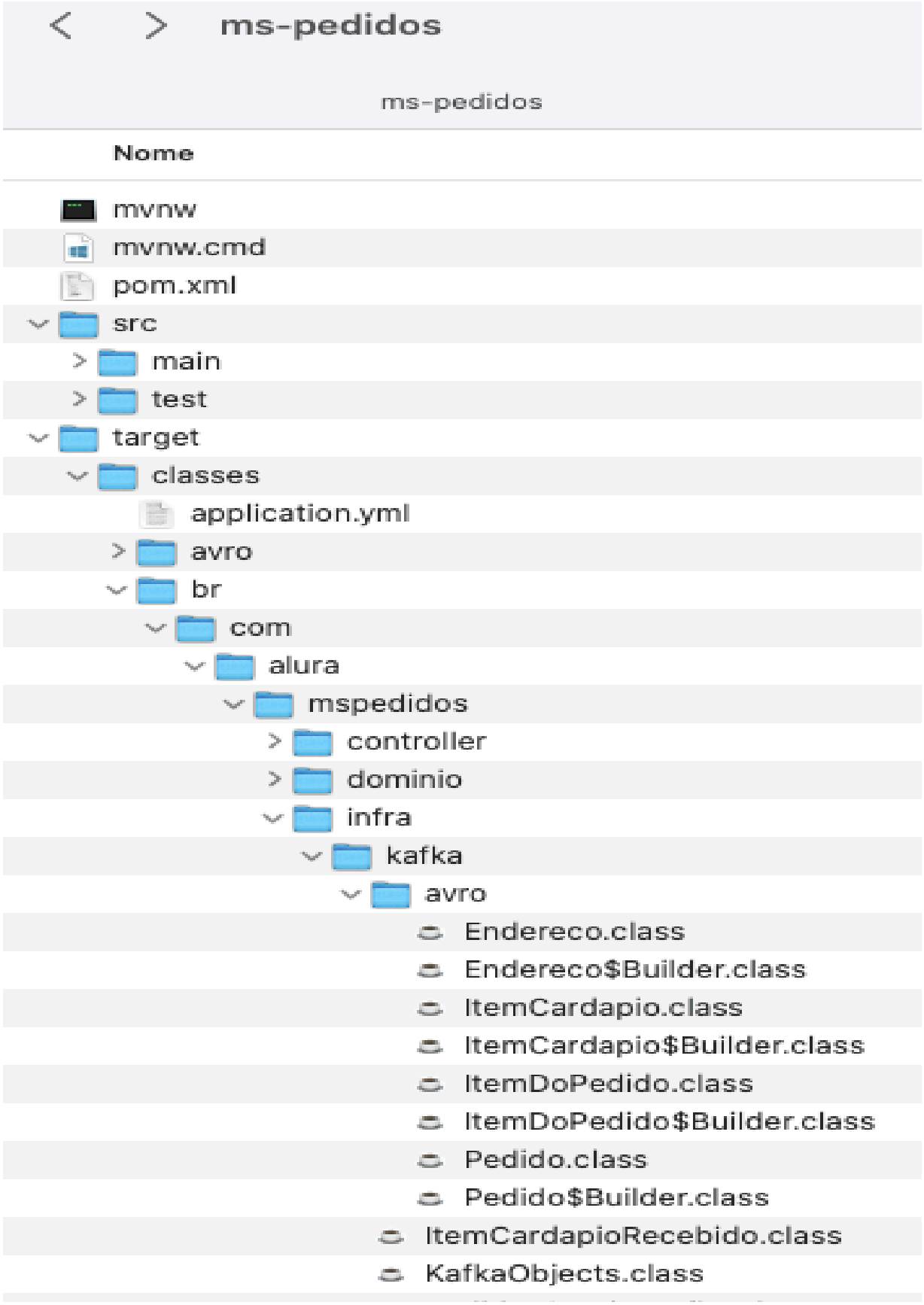

ms-pedidos
ms-pedidos
Nome
mvnw
mvnw.cmd
pom.xml
src
main
test
target
classes
application.yml
avro
br
com
alura
mspedidos
controller
dominio
infra
kafka
avro
Endereco.class
Endereco$Builder.class
ItemCardapio.class
ItemCardapio$Builder.class
ItemDoPedido.class
ItemDoPedido$Builder.class
Pedido.class
Pedido$Builder.class
ItemCardapioRecebido.class
KafkaObjects.class

Figura 5.3: Classes geradas por meio do avro-maven-plugin.

Podemos inclusive ver como elas foram geradas. Posteriormente, o exemplo da classe `Endereco.class`:

```
package br.com.alura.mspedidos.infra.kafka.avro;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.nio.ByteBuffer;
import org.apache.avro.AvroMissingFieldException;
import org.apache.avro.AvroRuntimeException;
import org.apache.avro.Schema;
import org.apache.avro.data.RecordBuilder;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.io.Encoder;
import org.apache.avro.io.ResolvingDecoder;
import org.apache.avro.message.BinaryMessageDecoder;
import org.apache.avro.message.BinaryMessageEncoder;
import org.apache.avro.message.SchemaStore;
import org.apache.avro.specific.AvroGenerated;
import org.apache.avro.specific.SpecificData;
import org.apache.avro.specific.SpecificRecord;
import org.apache.avro.specific.SpecificRecordBase;
import
org.apache.avro.specific.SpecificRecordBuilderBase;
import org.apache.avro.util.Utf8;

@AvroGenerated
public class Endereco extends SpecificRecordBase
implements SpecificRecord {
    private static final long serialVersionUID =
-5452243541848842594L;
    public static final Schema SCHEMA$ = (new
```

```
Schema.Parser()).parse("
{\"type\":\"record\",\"name\":\"Endereco\",\"namespace\
":\"br.com.alura.mspedidos.infra.kafka.avro\",\"doc\":\
"O endereço onde o pedido deve ser
entregue\",\"fields\":
[{\"name\":\"rua\",\"type\":\"string\"},
{\"name\":\"numero\",\"type\":\"int\"},
{\"name\":\"bairro\",\"type\":\"string\"},
{\"name\":\"cidade\",\"type\":\"string\"},
{\"name\":\"estado\",\"type\":\"string\"},
{\"name\":\"cep\",\"type\":\"string\"},
{\"name\":\"complemento\",\"type\":
[\"null\",\"string\"],\"default\":null}]}");
    private static final SpecificData MODEL$ = new
SpecificData();
    private static final BinaryMessageEncoder<Endereco>
ENCODER;
    private static final BinaryMessageDecoder<Endereco>
DECODER;
    private CharSequence rua;
    private int numero;
    private CharSequence bairro;
    private CharSequence cidade;
    private CharSequence estado;
    private CharSequence cep;
    private CharSequence complemento;
    private static final DatumWriter<Endereco> WRITER$;
    private static final DatumReader<Endereco> READER$;

    public static Schema getClassSchema() {
        return SCHEMA$;
    }

    public static BinaryMessageEncoder<Endereco>
getEncoder() {
        return ENCODER;
```

```
    }

    public static BinaryMessageDecoder<Endereco>
getDecoder() {
        return DECODER;
    }

    public static BinaryMessageDecoder<Endereco>
createDecoder(SchemaStore resolver) {
        return new BinaryMessageDecoder(MODEL$,
SCHEMA$, resolver);
    }

    public ByteBuffer toByteBuffer() throws IOException
{
        return ENCODER.encode(this);
    }

    public static Endereco fromByteBuffer(ByteBuffer b)
throws IOException {
        return (Endereco)DECODER.decode(b);
    }

    public Endereco() {
    }

    public Endereco(CharSequence rua, Integer numero,
CharSequence bairro, CharSequence cidade, CharSequence
estado, CharSequence cep, CharSequence complemento) {
        this.rua = rua;
        this.numero = numero;
        this.bairro = bairro;
        this.cidade = cidade;
        this.estado = estado;
        this.cep = cep;
```

```
        this.complemento = complemento;
    }
```

Embora seja um arquivo com extensão `.class` (ou seja, uma classe Java compilada) é possível visualizar o seu código-fonte. IDEs como IntelliJ ou Eclipse fornecem esse recurso.

Note que ela:

- estende `org.apache.avro.specific.SpecificRecordBase`
- e implementa `import org.apache.avro.specific.SpecificRecord`

Isso é necessário porque, com elas, os objetos Java são convertidos em mensagens no formato Avro.

5.5 O protocolo SOAP (Simple Object Access Protocol)

Há algum tempo, quando precisávamos integrar duas aplicações via HTTP, usávamos, em vez do REST, o protocolo SOAP. Esse protocolo tinha como uma das principais características, ainda que não obrigatória, o envio de mensagens no formato XML.

Para que isso fosse possível, havia diversos *frameworks* (JAXB, Axis, Apache CXF) que

convertiam objetos Java para XML (ou vice-versa), facilitando a programação dessas integrações. Essas aplicações geravam um arquivo WSDL (*Web Service Description Language*) e, com ele, os *frameworks* geravam os *stubs* Java.

O `avro-maven-plugin` baseia-se no mesmo princípio: por meio de bibliotecas adequadas, geram-se os *stubs* a partir dos arquivos AVSC. Com isso, o programador ou a programadora não precisa se preocupar com os detalhes do protocolo Avro: basta que gere as classes por meio do *plugin*.

Com os *stubs* gerados, podemos então cadastrar o *schema* no Schema Registry. Veremos a seguir algumas maneiras de fazer isso.

5.6 Cadastro de schemas no Schema Registry

Uma vez que os *schemas* estejam criados no produtor, é possível cadastrá-los no Schema Registry. Assim, o tópico poderá utilizá-los para validar as mensagens enviadas.

Esse cadastro pode ser realizado de algumas maneiras:

- cadastro de cada um dos *schemas* criados no Schema Registry. Nesse cenário, há um reaproveitamento de *schemas* por mais de um tópico. O *schema* de `Endereco`, por exemplo, pode ser reutilizado por diversos tópicos cujas mensagens possuam essa estrutura de dados.
- criação de um *schema* único para cadastro no Schema Registry, destinado ao tópico `pedidos`. Essa maneira impede reaproveitamento de *schemas* por outros tópicos. Em compensação, evita uma possível dependência entre *schemas*, pois, caso um dos *schemas* seja alterado, todos os tópicos dependentes desse *schema* serão afetados.

Um schema por entidade

No primeiro caso, podemos usar o Redpanda Console para cadastrar cada um dos *schemas*, iniciando com o cadastro do *schema* de `Endereco`:

- acesse o Schema Registry e clique no botão "Create new schema";
- em "Strategy", selecionar "Record";
- em "Key or value", selecionar "Value";
- em "Schema definition", selecionar o formato "Avro";

- copiar o *schema* `Endereco.avsc` e colar no Redpanda Console;
- clicar no botão abaixo "Validate" para que o Redpanda Console verifique se está tudo certo com o *schema* copiado;
- se tudo estiver correto, clicar no botão "Save". O *schema* será cadastrado e poderá ser usado para validar diversas mensagens de outros tópicos do Kafka, não somente o tópico `pedidos`.

Repita o processo para os *scripts* `ItemCardapio.avsc`, `ItemDoPedido.avsc` e `Pedido.avsc`, com algumas diferenças. Como `ItemDoPedido.avsc` depende de `ItemCardapio.avsc`, devemos informar essa dependência no Redpanda Console. Para isso:

- no cadastro de `ItemDoPedido.avsc`, em "Schema reference", informar `itemCardapio`. Informe `ItemCardapio-value` em "subject".
- validar, clicando em "Validate" e, em caso positivo, clicar em "Save". O *schema* `ItemDoPedido.avsc` estará cadastrado no Schema Registry:

Abaixo, a inserção do *schema* no Redpanda:

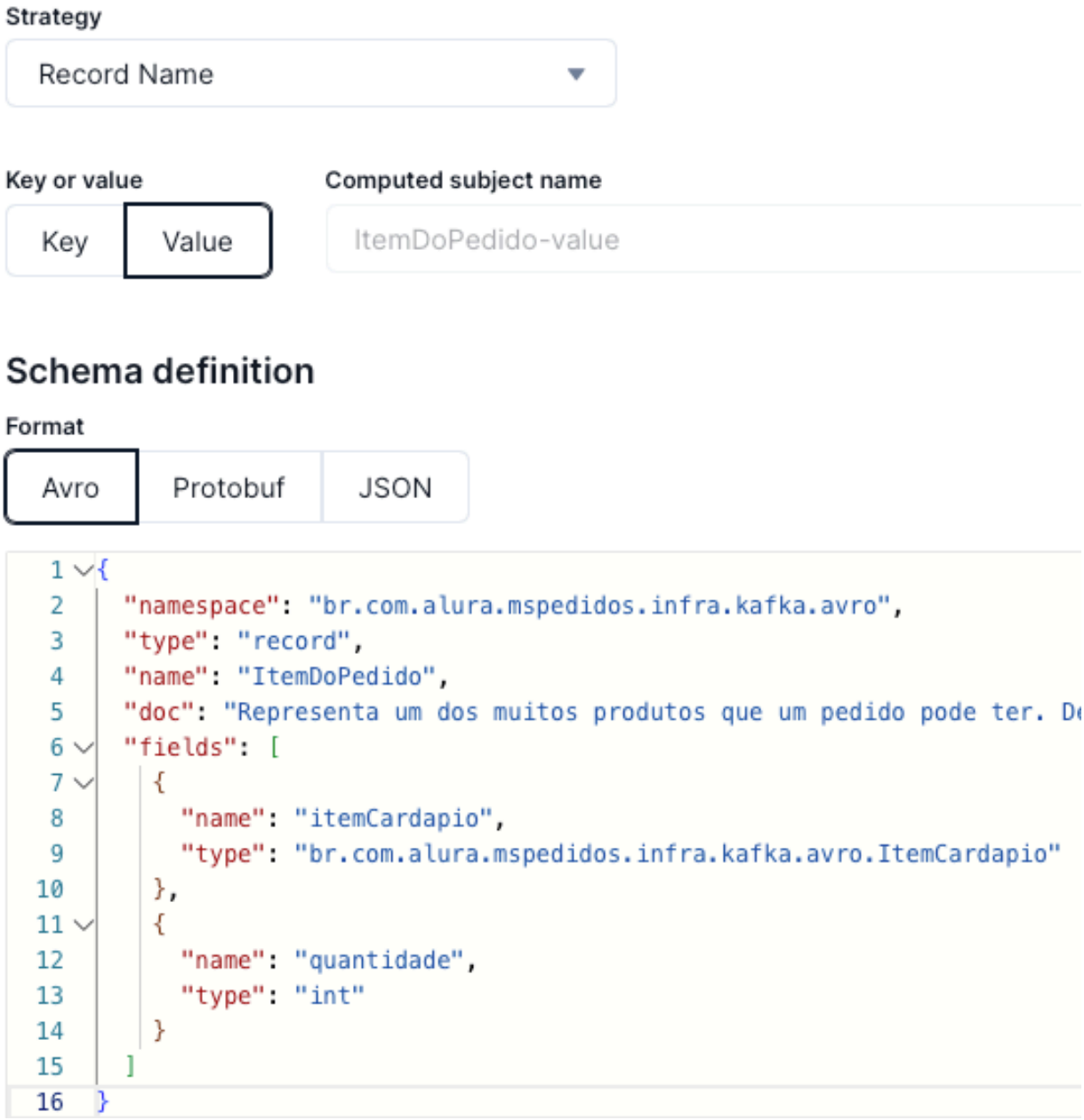


Figura 5.4: Cadastro do schema ItemDoPedido.avsc.

E, em seguida, o cadastro da dependência entre `ItemDoPedido.avsc` e `ItemCardapio.avsc`:

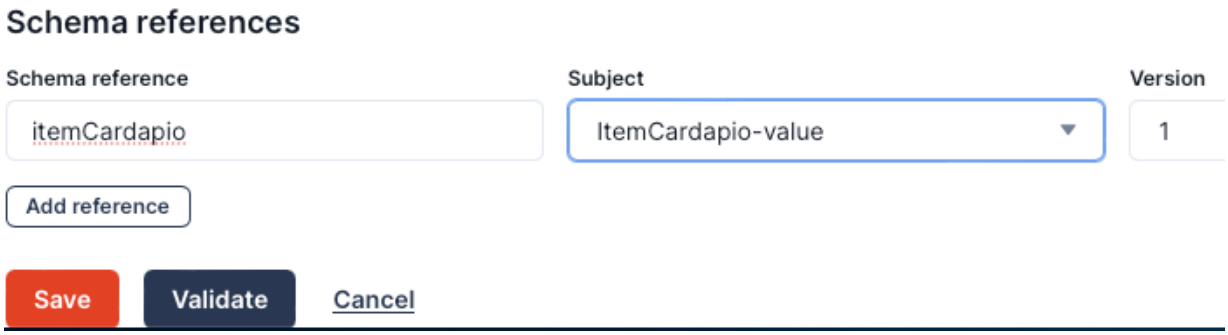


Figura 5.5: Associação entre ItemDoPedido.avsc e ItemCardapio.avsc.

Para o `Pedido.avsc`, podemos cadastrar o *schema* associado diretamente ao tópico `pedidos`:

- selecione o tópico em "Topic name", no cadastro de *schemas*
- escolha "Value" em "Key or value"

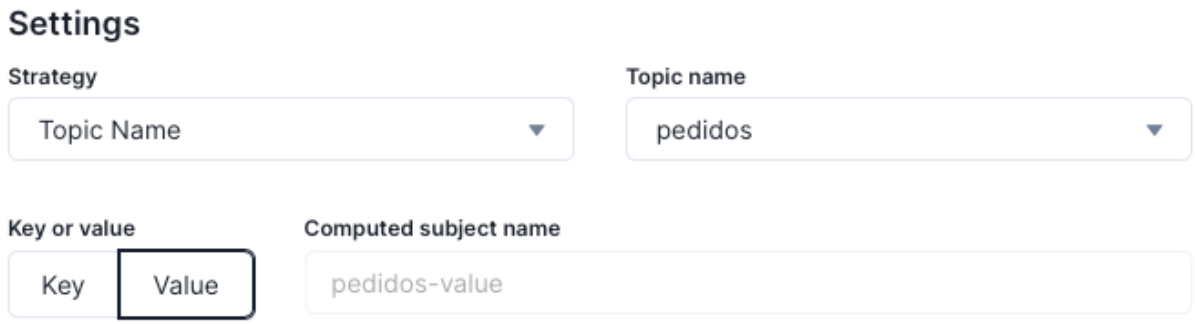


Figura 5.6: Seleção de tópico e de valor.

- copie o *schema* `Pedido.avsc` para o "Schema definition" (selecionando o formato Avro)

Schema definition

Format

Avro | Protobuf | JSON

```
{
  "namespace": "br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "name": "Pedido",
  "doc": "Representa uma compra de produtos de um mesmo restaurante",
  "fields": [
    {
      "name": "itensDoPedido",
      "type": {
        "type": "array",
        "items": {
          "type": "br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedido"
        }
      }
    },
    {
      "name": "endereco",
      "type": "br.com.alura.mspedidos.infra.kafka.avro.Endereco"
    }
  ]
}
```

Figura 5.7: Inserção do schema Pedido.avsc.

- e referencie os *schemas* de endereço e de item de pedido:

Schema references

Schema reference	Subject	Version
endereco	Endereco-value	1
itemDoPedido	ItemDoPedido-value	1

Figura 5.8: Cadastro do schema Pedido.avsc.

Com isso, podemos enviar mensagens para o tópico `pedidos` usando o *ms-pedidos*. Basta que criemos um novo pedido usando o cURL adiante:

```
curl --location 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o
limite"
      },
      "quantidade": 2
    }
  ],
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
'
```

Teremos um novo pedido no tópico de `pedidos`:

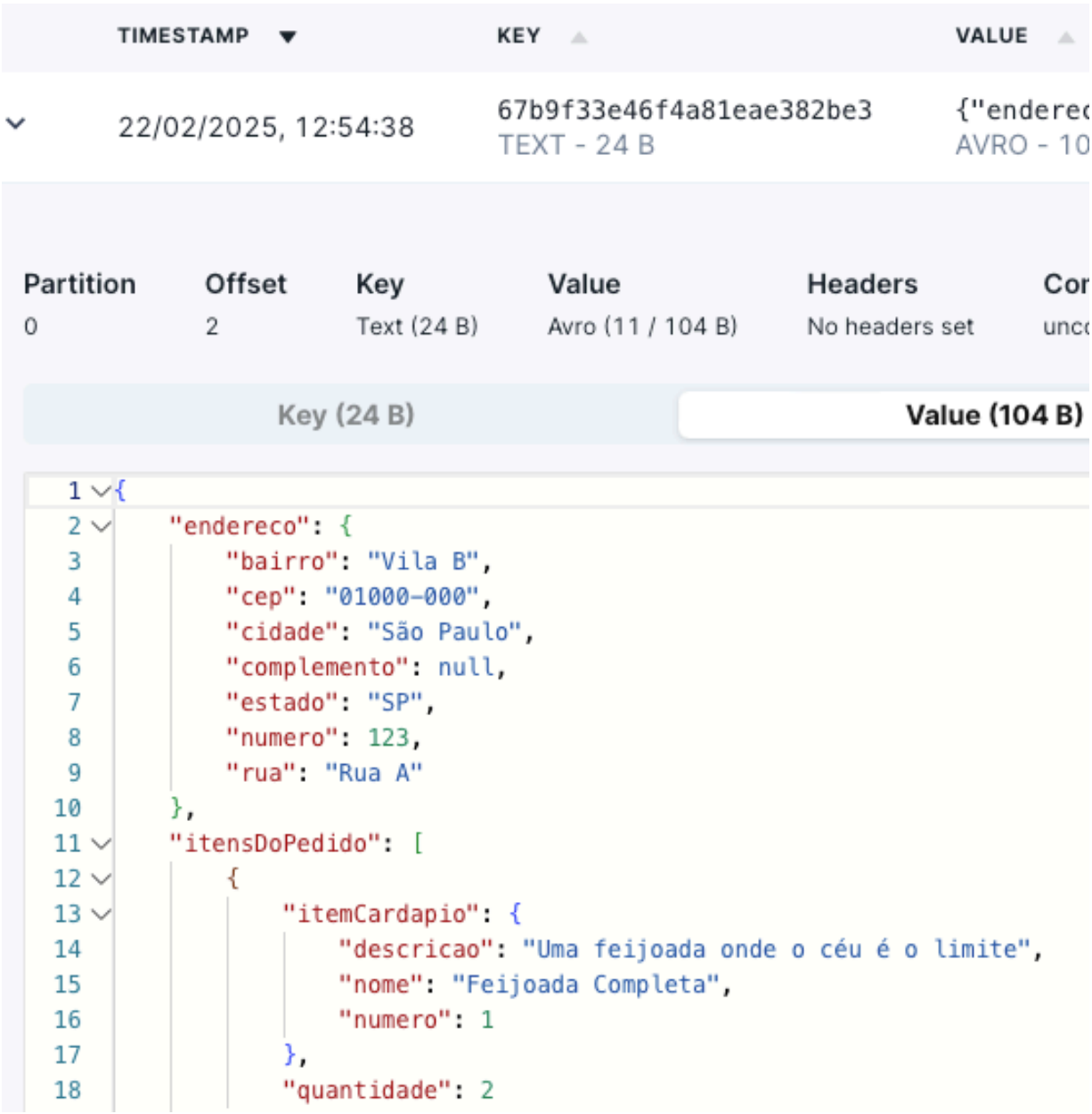


Figura 5.9: Novo pedido gerado.

Essa é uma possível maneira de cadastrar os *schemas* necessários no Schema Registry. Entretanto, há algumas limitações:

- Alterações em algum *schema* devem levar em consideração que outros tópicos podem utilizar o *schema* a ser alterado, o que aumenta a complexidade de configuração e gestão.
- Para entender a mensagem presente em um tópico, é necessário o entendimento de mais de um *schema*.
- Para cada busca de um *schema* (seja para produção ou consumo de mensagens) pode ser necessária uma requisição ao Schema Registry, gerando uma latência de rede adicional.

Um schema único por tópico - o Avro Schema Merger

Nessa maneira, basicamente cria-se um *schema* a partir da junção dos *schemas* que formam a mensagem, gerando um novo arquivo AVSC.

Pode-se gerar esse arquivo manualmente, copiando e colando os *schemas* para um arquivo novo, o que pode ser trabalhoso e propenso a erros. Ou então podemos utilizar ferramentas que realizam essa junção automaticamente. Uma delas é o Avro Schema Merger, criado especialmente para este livro.

O Avro Schema Merger é uma aplicação simples em Java que recebe como parâmetros:

- a pasta raiz de todos os *scripts*;
- cada um dos *scripts*, na mesma ordem que eles foram declarados no `pom.xml`.

E retorna no console o *script* único, composto por todos os outros. Sua classe principal está na sequência. Os comentários buscam explicar o que está sendo feito em cada ponto do processamento:

```
package br.com.alura.avroschemamerger;

import org.apache.avro.Schema;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

public class AvroSchemaMerger {

        public final String merge(File pastaRaiz,
boolean comSchema, List<String> arquivosDaPastaEmOrdem)
throws IOException {
        List<File> schemas =
arquivosDaPastaEmOrdem.stream()
                .map(nomeArquivoAvsc -> new
File(pastaRaiz, nomeArquivoAvsc))
                .toList();

        if (schemas.isEmpty()) {
            System.out.println("Nenhum arquivo .avsc
encontrado.");
            return "";
        }
```

```
        // se contém apenas 1 arquivo, não é necessário
realizar merge. Apenas adicione o conteúdo
        // dentro do schema
        if(schemas.size() == 1){
            ObjectMapper mapper = new ObjectMapper();
            ObjectNode rootNode =
mapper.createObjectNode();
            rootNode.put("schema", new
String(Files.readAllBytes(schemas.getFirst().toPath()))
);

            return mapper.writeValueAsString(rootNode);
        }

        // Criar um único parser para registrar os
schemas
        Schema.Parser parser = new Schema.Parser();

        // Primeiro, carregamos todos os schemas e os
registramos no parser
        for (File schemaFile : schemas) {
            parser.parse(new
String(Files.readAllBytes(schemaFile.toPath())));
        }

        //depois obtemos o último parser, visto que ele
é o raiz, e o schema
        String chave = parser.getTypes().keySet()
                .stream().reduce((primeiro, segundo) ->
segundo).orElse(null);

        String schemaMergeadoJson =
parser.getTypes().get(chave).toString(true);

        if(comSchema){
```

```
            ObjectMapper mapper = new ObjectMapper();
            ObjectNode rootNode =
mapper.createObjectNode();
            rootNode.put("schema", schemaMergeadoJson);

            return mapper.writeValueAsString(rootNode);
        }
        return schemaMergeadoJson;
    }
}
```

O Avro Schema Merger também está disponível na *branch* cap-5. Acesse-o para mais detalhes de sua implementação, inclusive os testes unitários escritos.

Para facilitar a criação desse *script*, deve-se gerar o pacote `jar` da aplicação (via `mvn package`) e executar o comando a seguir:

```
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-pedidos/src/main/avro true Endereco.avsc
ItemCardapio.avsc ItemDoPedido.avsc Pedido.avsc
```

Ao rodá-lo, obtém-se o seguinte resultado:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo
avroschemamerger % java -jar target/avro-schema-merger-
0.0.1-SNAPSHOT.jar ../ms-pedidos/src/main/avro true
Endereco.avsc ItemCardapio.avsc ItemDoPedido.avsc
Pedido.avsc
{
  "type" : "record",
```

```
  "name" : "Pedido",
  "namespace" :
"br.com.alura.mspedidos.infra.kafka.avro",
  "doc" : "Representa uma compra de produtos de um
mesmo restaurante",
  "fields" : [ {
    "name" : "itensDoPedido",
    "type" : {
      "type" : "array",
      "items" : {
        "type" : "record",
        "name" : "ItemDoPedido",
        "doc" : "Representa um dos muitos produtos que
um pedido pode ter. Descreve também a quantidade de
produtos de mesmo tipo",
        "fields" : [ {
          "name" : "itemCardapio",
          "type" : {
            "type" : "record",
            "name" : "ItemCardapio",
            "doc" : "Descreve um item do cardápio.
Usado para descrever detalhadamente um item de pedido",
            "fields" : [ {
              "name" : "numero",
              "type" : "long"
            }, {
              "name" : "nome",
              "type" : "string"
            }, {
              "name" : "descricao",
              "type" : "string"
            } ]
          }
        }, {
          "name" : "quantidade",
          "type" : "int"
```

```
        } ]
      }
    }
  }, {
    "name" : "endereco",
    "type" : {
      "type" : "record",
      "name" : "Endereco",
      "doc" : "O endereço onde o pedido deve ser
entregue",
      "fields" : [ {
        "name" : "rua",
        "type" : "string"
      }, {
        "name" : "numero",
        "type" : "int"
      }, {
        "name" : "bairro",
        "type" : "string"
      }, {
        "name" : "cidade",
        "type" : "string"
      }, {
        "name" : "estado",
        "type" : "string"
      }, {
        "name" : "cep",
        "type" : "string"
      }, {
        "name" : "complemento",
        "type" : [ "null", "string" ],
        "default" : null
      } ]
    }
  } ]
}
```

O *schema* resultante está devidamente formatado para facilitar a visualização.

Para cadastrarmos esse novo *schema*, devemos inicialmente apagar os *scripts* que criamos anteriormente. Podemos usar para isso o Redpanda Console, a interface REST do Schema Registry (que já vimos no capítulo 3) ou mesmo reiniciar a infraestrutura de nossa solução, por meio do seguinte comando:

```
./infra-down.sh && ./infra-up.sh
```

Implementação do `PedidosSenderKafka`

Agora que cadastramos o *schema* no Schema Registry, podemos finalmente escrever a classe que implementa a interface `PedidosSender`.

Inicialmente, precisamos construir um *bean* do tipo `KafkaTemplate`, que será usado pelo `PedidosSenderKafka`. Podemos nos basear no que fizemos no *ms-restaurantes*:

- criamos uma classe `KafkaObjects` apenas como uma fábrica desses objetos necessários ao Spring;

- implementamos o método fábrica de `KafkaTemplate`.

Talvez a grande mudança está no método `produtorConfigs()`:

- para serializar a chave, usamos a classe `StringSerializer`;
- para serializar o valor, usamos `KafkaAvroSerializer`.

O código de `KafkaObjects` está a seguir (trechos omitidos por brevidade):

```
@Value("${spring.kafka.bootstrap-servers}")
private String kafkaServerUrl;

@Value("${spring.kafka.schema.registry.url}")
private String schemaRegistryUrl;

@Bean
KafkaTemplate<String, Pedido> kafkaTemplate(){
    return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(produtorConfigs()));
}

private Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
```

```
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);


props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

    return props;
}
```

Já a classe `PedidosSenderKafka` não é muito diferente de `ItensCardapioSenderKafka`. Veja adiante:

```
@Component
class PedidosSenderKafka implements PedidosSender {

    private final KafkaTemplate<String,
br.com.alura.mspedidos.infra.kafka.avro.Pedido>
kafkaTemplate;
    private final String topico;

    PedidosSenderKafka(KafkaTemplate<String,
br.com.alura.mspedidos.infra.kafka.avro.Pedido>
kafkaTemplate,
                       @Value("${topico.pedidos}")
String topico) {
        this.kafkaTemplate = kafkaTemplate;
        this.topico = topico;
    }
```

```
    @Override
    public void enviar(Pedido pedido) {
        kafkaTemplate.send(topico, pedido.getId(),
pedido.toAvro());
    }
}
```

Observe que, no método `enviar()`:

- recebemos `br.com.alura.mspedidos.dominio.Pedido`
- e enviamos para o método `send()` de `KafkaTemplate` um `br.com.alura.mspedidos.infra.kafka.avro.Pedido`

Essa conversão é feita usando método `toAvro()` da classe `br.com.alura.mspedidos.dominio.Pedido`, descrito a seguir:

```
public br.com.alura.mspedidos.infra.kafka.avro.Pedido
toAvro() {
    return
br.com.alura.mspedidos.infra.kafka.avro.Pedido.newBuild
er()
            .setEndereco(endereco.toAvro())

.setItensDoPedido(itensDoPedido.stream().map(ItemDoPedi
do::toAvro).collect(toList()))
            .build();
}
```

É possível verificar que existem métodos semelhantes a ele nas classes `Endereco` e `ItemDoPedido`. Tanto na classe `Pedido` quanto nas demais, os atributos são enviados para instâncias dos *stubs* criados.

> Essa maneira de criar objetos é baseada no padrão de projeto *builder*, que permite a criação de projetos de forma mais maleável se comparada à construção baseada em construtores. Para mais informações sobre esse padrão, uma excelente referência é o livro *Padrões de projetos: Soluções reutilizáveis de software orientado a objetos* (2006).

Testes de envio de mensagens

Assim como fizemos nas demais classes dos nossos microsserviços, vamos testar a classe `PedidosSenderKafkaTest`. Já criamos testes de envio de mensagens para o `ItensCardapioSenderKafka`. E, felizmente, o teste para o `PedidosSenderKafka` não será muito diferente:

- na classe `KafkaObjectsParaTestes`, exibida a seguir e escrita em *ms-pedidos*, implementamos o `bean`

de consumo de mensagens, que verificará se a mensagem enviada pelo `PedidosSenderKafka` chegou com sucesso ao tópico. O método `configsDoConsumidor()` retorna o mapa de configurações do consumidor do teste. E nele passamos a trabalhar com os seguintes desserializadores:

- `StringDeserializer` para desserializar a chave.
- `KafkaAvroDeserializer` para desserializar o valor.

```
// trechos de código omitidos por brevidade

@Value("${spring.kafka.bootstrap-servers}")
private String kafkaServerUrl;

@Value("${spring.kafka.schema.registry.url}")
private String schemaRegistryUrl;

@Value("${topico.pedidos}")
private String topicoDePedidos;

@Value("${spring.kafka.consumer.group-id}")
private String grupo;

@Value("${spring.kafka.consumer.auto-offset-reset}")
private String autoOffsetReset;

@Bean
KafkaConsumer<String, Pedido> consumidorDeMensagens(){
    KafkaConsumer<String, Pedido> consumidor =
            new KafkaConsumer<>(configsDoConsumidor());
```

```
consumidor.subscribe(singletonList(topicoDePedidos));
    return consumidor;
}

public Map<String, Object> configsDoConsumidor() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
    );
}
```

A classe `PedidosSenderKafkaTest` é praticamente igual a `ItensCardapioSenderKafkaTest`:

```
package br.com.alura.mspedidos.infra.kafka;

import br.com.alura.mspedidos.infra.kafka.avro.Pedido;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.awaitility.core.ThrowingRunnable;
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowired;
import
```

```
org.springframework.boot.test.context.SpringBootTest;
import
org.springframework.kafka.test.context.EmbeddedKafka;

import java.time.Duration;
import java.util.HashMap;
import java.util.Map;

import static
br.com.alura.mspedidos.dominio.ObjetosDeDominio.umPedid
o;
import static
org.assertj.core.api.Assertions.assertThat;
import static org.awaitility.Awaitility.await;

@SpringBootTest
@EmbeddedKafka
class PedidosSenderKafkaTest {

    @Autowired
    private PedidosSenderKafka pedidosSenderKafka;

    @Autowired
    private KafkaConsumer<String, Pedido> consumidor;

    @Test
    void enviarNovoPedido(){
        var pedido = umPedido();
        pedidosSenderKafka.enviar(pedido);

        esperarUmTempoAte(() ->
                assertThat(mensagensDoTopico())
                        .containsEntry(pedido.getId(),
pedido.toAvro()));
    }
```

```
    private Map<String, Pedido> mensagensDoTopico(){
        Map<String, Pedido> eventos = new HashMap<>();


consumidor.poll(Duration.ofSeconds(1)).forEach(consumer
Record ->
                eventos.put(consumerRecord.key(),
consumerRecord.value())
        );

        return eventos;
    }

    private void esperarUmTempoAte(ThrowingRunnable
runnable){
        await()
                .atMost(Duration.ofSeconds(10))
                .untilAsserted(runnable);
    }
}
```

Podemos, inclusive, rodar o teste:

```
mvn -f ms-pedidos/pom.xml clean test -
Dtest=PedidosSenderKafkaTest
```

Entretanto, o seguinte erro ocorre:

```
[ERROR] Tests run: 1, Failures: 0, Errors: 1, Skipped:
0, Time elapsed: 20.20 s <<< FAILURE! -- in
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st
[ERROR]
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st.enviarNovoPedido -- Time elapsed: 4.341 s <<< ERROR!
java.lang.ClassCastException: class
```

```
org.apache.avro.generic.GenericData$Record cannot be
cast to class
br.com.alura.mspedidos.infra.kafka.avro.Pedido
(org.apache.avro.generic.GenericData$Record and
br.com.alura.mspedidos.infra.kafka.avro.Pedido are in
unnamed module of loader 'app')
    at
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st.lambda$mensagensDoTopico$1(PedidosSenderKafkaTest.ja
va:43)
    at
java.base/java.lang.Iterable.forEach(Iterable.java:75)
    at
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st.mensagensDoTopico(PedidosSenderKafkaTest.java:42)
    at
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st.lambda$enviarNovoPedido$0(PedidosSenderKafkaTest.jav
a:35)
    at
org.awaitility.core.AssertionCondition.lambda$new$0(Ass
ertionCondition.java:53)
    at
org.awaitility.core.ConditionAwaiter$ConditionPoller.ca
ll(ConditionAwaiter.java:248)
```

Repare que estranhamente ocorre uma conversão inesperada:

- de `org.apache.avro.generic.GenericData$Record`
- para

  `br.com.alura.mspedidos.infra.kafka.avro.Pedido`

Por que isso acontece se estamos trabalhando o tempo todo com a mensagem no formato `br.com.alura.mspedidos.infra.kafka.avro.Pedido` e não com esse `GenericData$Record` ? Explicaremos a seguir.

Parâmetro `KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG`

Basicamente, no nosso teste, estamos usando a classe `KafkaAvroDeserializer` para desserializar a mensagem para objeto Java. Acontece que, por padrão, esse desserializador trabalha de forma genérica, retornando um objeto que possa lidar com todas as mensagens possíveis de serem criadas. Esse objeto é o `org.apache.avro.generic.GenericRecord`.

Se queremos trabalhar com a desserialização automática para os objetos que especificamos no teste, precisamos informar mais um parâmetro no método `configsDoConsumidor()` de `KafkaObjectsParaTestes`: o `KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG`. Esse parâmetro aceita valores do tipo `boolean`, e

se não o configurarmos, utiliza-se o seu valor padrão, que é `false`.

Logo, precisamos incluir esse parâmetro no mapa presente no método `configsDoConsumidor()`:

```
public Map<String, Object> configsDoConsumidor() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
            ,
KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG
, true
    );
}
```

Agora, conseguimos rodar o teste do produtor de mensagens sem problemas:

```
mvn -f ms-pedidos/pom.xml clean test -
Dtest=PedidosSenderKafkaTest

# trechos omitidos por brevidade
```

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 29.55 s -- in
br.com.alura.mspedidos.infra.kafka.PedidosSenderKafkaTe
st
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
------------------------
```

Com o produtor de mensagens escrito e testado, subimos o *ms-pedidos* e já podemos enviar novos pedidos, usando o Postman ou o cURL:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl
--location 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o
limite"
      },
      "quantidade": 2
    }
  ],
```

```
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
'
```

Eles devem chegar normalmente ao tópico:

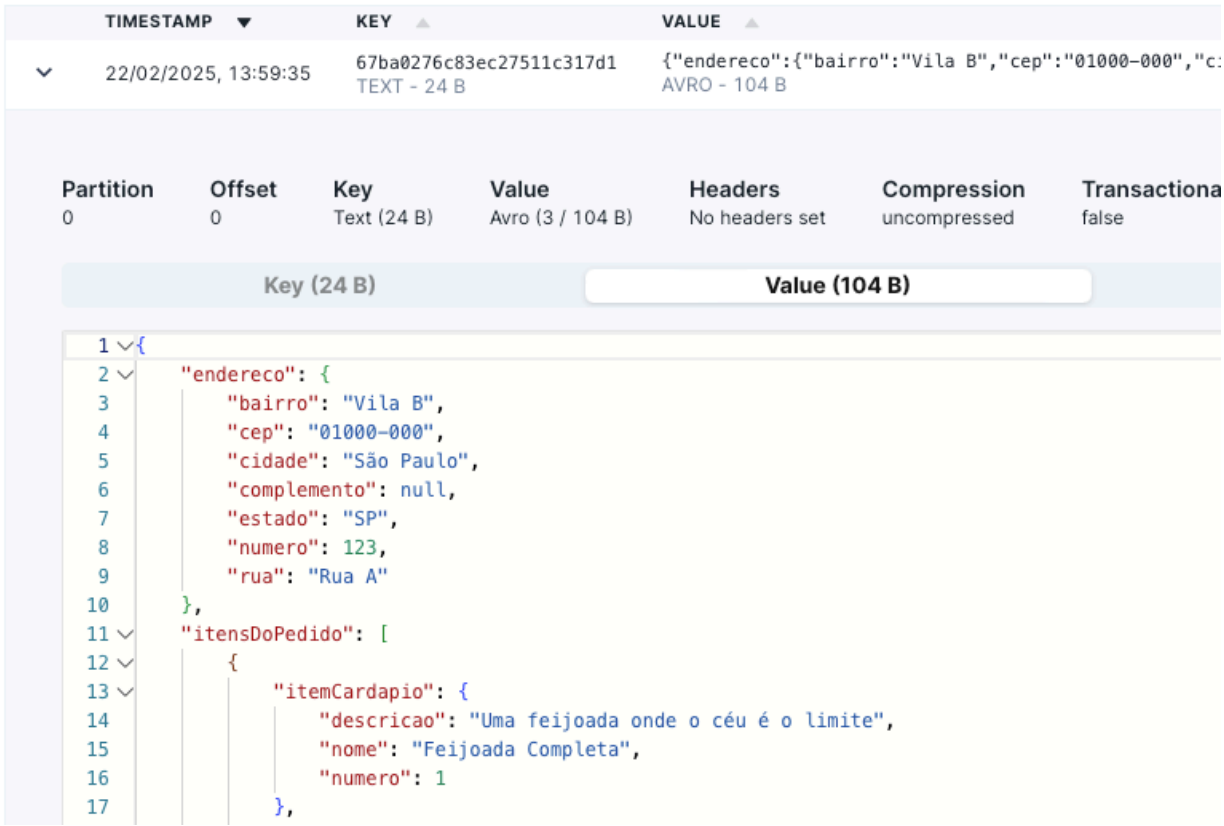


Figura 5.10: Pedido gerado após o cadastro do schema único para o tópico.

5.7 Resumo

Neste capítulo, exploramos o protocolo Avro em comparação ao JSON. Também vimos como preparar nossas aplicações para o envio de mensagens para o Kafka usando esse protocolo, bem como o formato AVSC para criação de *schemas* e maneiras de cadastrar *schemas* Avro no Schema Registry e testes do produtor de mensagens.

No próximo capítulo, vamos compreender como consumir mensagens nesse formato. Também usaremos o formato AVDL para geração de mensagens Avro.

CAPÍTULO 6

Consumidores de mensagens Avro e o AVDL

No capítulo anterior, vimos algumas definições sobre o protocolo Avro e suas vantagens em relação ao JSON. Também implementamos o envio de pedidos para o Kafka utilizando esse protocolo.

Neste capítulo, vamos implementar um consumidor dessas mensagens. Veremos também outro formato de definição de *schema*: o AVDL, que significa Avro IDL (*Interface Definition Language*).

> Os códigos deste capítulo encontram-se na *branch* `cap-6`.

Mas antes de fazer a implementação, vamos realizar algumas melhorias o código do capítulo anterior para facilitar o entendimento.

6.1 Automação do cadastro do schema do tópico de pedidos

No final do capítulo anterior, usamos o Avro Schema Merger como uma ferramenta que facilita a criação de *schemas* únicos por tópico. Utilizamos essa mesma ferramenta para gerar o *schema* do tópico `pedidos`.

Vamos automatizar esse cadastro para que sua execução ocorra durante a subida da infraestrutura da solução, assim como fizemos com os demais *schemas* no capítulo 3.

Inicialmente, vamos criar um *script* que gere o artefato executável do Avro Schema Merger na raiz do projeto, caso ele não exista. Isso será o `instalar_avroschemamerger.sh`, descrito a seguir:

```
AVRO_SCHEMA_MERGER="target/avro-schema-merger-0.0.1-
SNAPSHOT.jar"
if [ ! -f "$AVRO_SCHEMA_MERGER" ]; then
  echo "Avro schema merger não encontrado. Gerando
pacote..."
  mvn clean package
fi
```

Agora podemos criar um *script* shell que faça o cadastro do *script* Avro no Schema Registry. Esse *script* deve estar na pasta `schemas` e seguir os seguintes passos:

- criar o executável do Avro Schema Merger, caso ele não exista;

- rodar o Avro Schema Merger, passando como parâmetro a pasta onde se encontram os *scripts* Avro e os nomes de cada um deles em ordem, bem como o arquivo de saída do resultado da execução (que deve ser também a pasta `schemas`);
- cadastrar o *schema* recém-criado no Schema Registry utilizando o *endpoint* apropriado.

O *script* que executa essas tarefas está adiante:

```
cd ../avroschemamerger
./instalar_avroschemamerger.sh
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-pedidos/src/main/avro true Endereco.avsc
ItemCardapio.avsc ItemDoPedido.avsc Pedido.avsc >
../schemas/pedidos-value.js

cd ../schemas
curl -X POST http://localhost:8081/subjects/pedidos-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@pedidos-value.js
```

Seguindo a convenção que adotamos para os demais *scripts* de cadastro de *schemas* no Schema Registry, esse *script* deve se chamar `pedidos-value-cadastrar.sh` e é invocado no `infra-up.sh` , como a seguir:

```
# trechos de código omitidos por brevidade

while true; do
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
    break
  fi
  sleep 5
done
```

6.2 Integração entre o ms-pedidos e o ms-entregadores

Uma vez que os pedidos são solicitados pelos clientes via *ms-pedidos* (e preparados pelos restaurantes), eles já podem ser entregues nos devidos endereços.

Essas entregas são feitas pelos entregadores. Eles são selecionados por uma aplicação conhecida como *ms-entregadores*, que consome o tópico `pedidos`. Os dados do entregador selecionado, juntamente do

pedido, são enviados para um tópico chamado de `entregadores`.

> De forma a facilitar a implementação, cada entregador realiza apenas 1 entrega por vez.

A interação entre essas aplicações e os tópicos pode ser visualizada na figura a seguir:

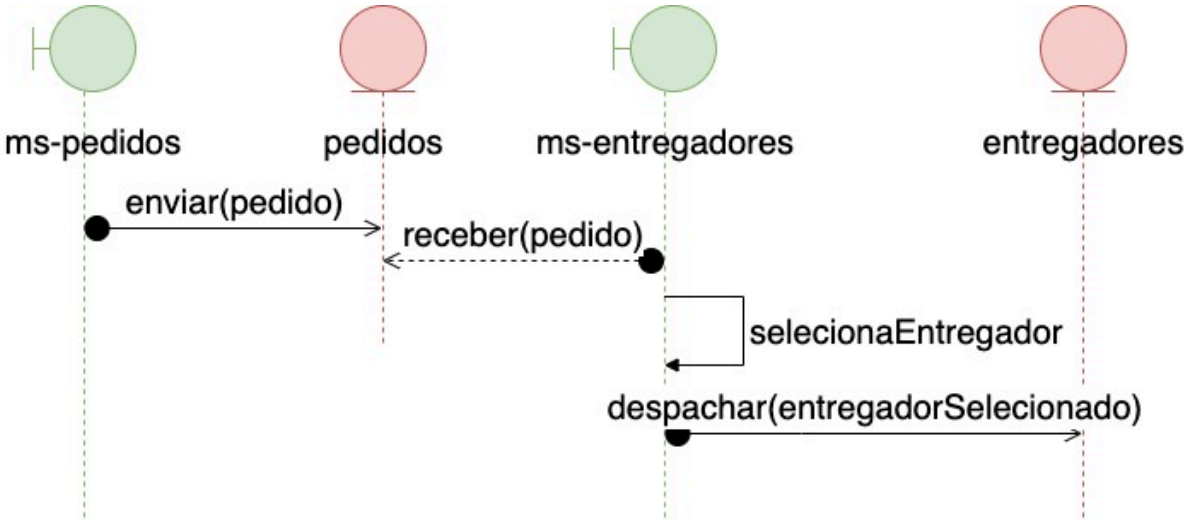


Figura 6.1: Integração entre o ms-pedidos e o ms-entregadores.

Para criarmos o tópico `entregadores`, vamos fazer o que já foi feito em capítulos anteriores: inserir um comando para sua criação em `docker-compose.yml`:

```
kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic
entregadores
```

Talvez a principal classe do *ms-entregadores*, do ponto de vista de negócio, seja a classe `Entregador`. Ela representa o profissional que fará as entregas dos pedidos. Possui métodos que permitem a verificação de seu estado atual, se podem ou não realizar entregas, além de poderem receber um `Pedido` para entrega. Seu código é o seguinte:

```
package br.com.alura.msentregadores.dominio;

import lombok.EqualsAndHashCode;

@EqualsAndHashCode(of = "nome")
public class Entregador {

    private final String nome;
    private EstadoEntregador status;

    public Entregador(String nome) {
        this.nome = nome;
        this.status = new Disponivel();
    }

    public final boolean podeEntregarOPedido() {
        return status.quePodeEntregarOPedido();
    }

    public void marcarEmEntrega(Pedido pedido) {
        status = new EmEntrega(pedido);
    }

    public void atualizar(Entregador entregador) {
        if(equals(entregador)) this.status =
entregador.status;
```

```
    }
}
```

Os possíveis estados de um `Entregador` são:

- `Disponivel` : indica que ele pode realizar a entrega de novos pedidos. Seu código encontra-se a seguir:

```
package br.com.alura.msentregadores.dominio;

public class Disponivel extends EstadoEntregador{

    @Override
    public boolean quePodeEntregarOPedido() {
        return true;
    }

    @Override
    public Pedido lerPedido() {
        return Pedido.NENHUM;
    }
}
```

- `EmEntrega` : indica que ele está entregando um pedido no momento. Seu código é bem parecido com o anterior:

```
package br.com.alura.msentregadores.dominio;

public class EmEntrega extends EstadoEntregador{

    private Pedido pedido;
```

```
    public EmEntrega(Pedido pedido){
        this.pedido = pedido;
    }

    @Override
    public boolean quePodeEntregarOPedido() {
        return false;
    }

    @Override
    public Pedido lerPedido() {
        return pedido;
    }
}
```

Esses estados foram construídos utilizando o padrão *State*, presente na lista de padrões de projeto do livro *Padrões de projetos: Soluções reutilizáveis de software orientado a objetos* (2006).

O *ms-entregadores* possui também um repositório simples em memória que armazena todos os entregadores, além de atualizá-los quando selecionados. Esse é chamado `TodosEntregadores` e possui alguns entregadores para que possamos usar a aplicação em alguns cenários. Veja seu código:

```
package br.com.alura.msentregadores.dominio;

import
```

```
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import java.util.Optional;
import java.util.Set;

@Repository
public class TodosEntregadores {

    private final Set<Entregador> entregadores;

    TodosEntregadores(Set<Entregador> entregadores){
        this.entregadores = entregadores;
    }

    @Autowired
    TodosEntregadores(){
        this(Set.of(
                new Entregador("Alice Stack"),
                new Entregador("Bruno Bit"),
                new Entregador("Carlos Kernel"),
                new Entregador("Daniel Buffer"),
                new Entregador("Eva Overflow"),
                new Entregador("Igor Pacote"),
                new Entregador("Juliana Script")
        ));
    }

    public Optional<Entregador> selecionarDisponivel()
{
        return entregadores.stream()
                .filter(entregador ->
entregador.podeEntregarOPedido())
                .findAny();
    }
```

```
    public void atualizar(Entregador entregador) {
        entregadores.stream()
                .filter(e -> e.equals(entregador))
                .findFirst().ifPresent(e ->
e.atualizar(entregador));
    }
}
```

Esse repositório é utilizado pela classe `SelecaoEntregadoresService`. Ela possui apenas um método, o `selecionarEntregadorPara(Pedido pedido)`, que recebe um pedido, busca um entregador disponível em `TodosEntregadores`, entrega o pedido ao entregador e o despacha para entrega. Seu código encontra-se a seguir:

```
package br.com.alura.msentregadores.service;

import br.com.alura.msentregadores.dominio.Despachante;
import br.com.alura.msentregadores.dominio.Pedido;
import
br.com.alura.msentregadores.dominio.TodosEntregadores;
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@AllArgsConstructor
public class SelecaoEntregadoresService {

    private TodosEntregadores todosEntregadores;
    private Despachante despachante;

    public void selecionarEntregadorPara(Pedido pedido)
{
```

```
todosEntregadores.selecionarDisponivel().ifPresent(entr
egador -> {
            entregador.marcarEmEntrega(pedido);
            todosEntregadores.atualizar(entregador);
            despachante.despachar(entregador);
        });
    }
}
```

A classe `Pedido` é uma classe de domínio, e busca representar o pedido para o negócio. Confira:

```
package br.com.alura.msentregadores.dominio;

import lombok.AccessLevel;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;

import java.util.Set;

import static
br.com.alura.msentregadores.dominio.Endereco.toEndereco
;
import static java.util.stream.Collectors.toList;

@NoArgsConstructor(access = AccessLevel.PRIVATE)
@EqualsAndHashCode
public class Pedido {

    private Set<ItemDoPedido> itensDoPedido;
    private Endereco endereco;

    public static final Pedido NENHUM = new Pedido();
}
```

Repare que ela não possui um método fábrica ou construtor que permita que ela seja construída com seus atributos devidamente valorados. Construiremos um método fábrica em breve.

Já o `Despachante` é uma interface a ser implementada na camada de infra. Possui apenas o método `despachar(Entregador entregador)`.

```
package br.com.alura.msentregadores.dominio;

public interface Despachante {

    void despachar(Entregador entregador);
}
```

Veremos sua implementação no final deste capítulo. Antes, precisamos ver o receptor de pedidos.

6.3 Geração de stubs Avro e implementação do consumidor de pedidos no ms-entregadores

Assim como geramos os *stubs* Avro no *ms-pedidos*, que era o produtor de mensagens para o tópico `pedidos`, devemos também gerá-los em todos os consumidores desse tópico (no caso, o *ms-entregadores*). Para isso, precisamos obter o *schema*

mais recente do *subject* `pedidos-value`, que cadastramos a partir da criação do produtor. Podemos:

- utilizar o Redpanda Console, acessando os *schemas* cadastrados;
- acessar o Schema Registry via interface REST, conforme vimos no capítulo 3, por meio do seguinte comando:

```
curl -X GET http://localhost:8081/subjects/pedidos-value/versions/latest/schema
```

Ao rodá-lo, temos como retorno:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl -X GET http://localhost:8081/subjects/pedidos-value/versions/latest/schema
{"type":"record","name":"Pedido","namespace":"br.com.alura.mspedidos.infra.kafka.avro","doc":"Representa uma compra de produtos de um mesmo restaurante","fields":[{"name":"itensDoPedido","type":{"type":"array","items":{"type":"record","name":"ItemDoPedido","doc":"Represent a um dos muitos produtos que um pedido pode ter. Descreve também a quantidade de produtos de mesmo tipo","fields":[{"name":"itemCardapio","type":{"type":"record","name":"ItemCardapio","doc":"Descreve um item do cardápio. Usado para descrever detalhadamente um item de pedido","fields":[{"name":"numero","type":"long"},{"name":"nome","type":"string"},{"name":"descricao","type":"string"}]}},{"name":"quantidade","type":"int"}]}}},
```

```
{"name":"endereco","type":
{"type":"record","name":"Endereco","doc":"O endereço
onde o pedido deve ser entregue","fields":
[{"name":"rua","type":"string"},
{"name":"numero","type":"int"},
{"name":"bairro","type":"string"},
{"name":"cidade","type":"string"},
{"name":"estado","type":"string"},
{"name":"cep","type":"string"},
{"name":"complemento","type":
["null","string"],"default":null}]}}]}%
```

Podemos então criar um arquivo em `src/main/avro` no *ms-entregadores*. Vamos chamá-lo de `Pedido.avsc`.

> Repare que o *schema* retornado pelo Schema Registry não é estruturalmente igual ao `Pedido.avsc` criado em *ms-pedidos*. Enquanto o `Pedido.avsc` faz referência a tipos nomeados, o *schema* retornado traz os tipos aninhados. De todo modo, os *stubs* serão criados sem problemas mais à frente.

Agora que já temos o *schema* a ser usado com o tópico, precisamos referenciar o `avro-maven-plugin` no `pom.xml`.

O código a ser inserido no `pom.xml` é muito semelhante ao já inserido no `pom.xml` de *ms-pedidos*.

A grande diferença entre eles é que no *ms-entregadores* precisamos referenciar apenas o `Pedido.avsc` que acabamos de criar:

```
<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>${avro.version}</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
            </goals>
            <configuration>
                <createSetters>false</createSetters>
                <imports>

<import>${project.basedir}/src/main/avro/Pedido.avsc</i
mport>
                </imports>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Ao rodarmos o comando `mvn -f ms-entregadores/pom.xml compile`, o *stub* `br.com.alura.mspedidos.infra.kafka.avro.Pedido` é gerado, e já podemos utilizá-lo no nosso código.

Com essa estrutura pronta, podemos finalmente pensar no receptor de pedidos. Felizmente, ele

também não terá muitas diferenças em relação ao receptor criado para o *ms-pedidos*:

- precisamos referenciar o tópico de pedidos no atributo `topics` da anotação `@KafkaListener`. Vamos chamar essa referência de `topico.pedidos`;
- para representarmos a mensagem, usaremos o *stub* criado.

O código será, portanto, o seguinte:

```
package br.com.alura.msentregadores.infra.kafka;

import
br.com.alura.msentregadores.service.SelecaoEntregadores
Service;
import br.com.alura.mspedidos.infra.kafka.avro.Pedido;
import lombok.AllArgsConstructor;
import
org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.KafkaHeaders;
import
org.springframework.messaging.handler.annotation.Header
;
import
org.springframework.messaging.handler.annotation.Payloa
d;
import org.springframework.stereotype.Component;

import static
br.com.alura.msentregadores.dominio.Pedido.fromMsPedido
s;

@Component
```

```
@AllArgsConstructor
class Receptor {

    private SelecaoEntregadoresService
selecaoEntregadores;

    @KafkaListener(topics = "${topico.pedidos}"
            , groupId = "${spring.kafka.consumer.group-
id}"
            , containerFactory = "kafkaListener")
    public void
receber(@Header(KafkaHeaders.RECEIVED_KEY) String
chave,
                        @Payload Pedido pedidoAvro){

selecaoEntregadores.selecionarEntregadorPara(fromMsPedi
dos(pedidoAvro));
    }
}
```

O método estático `fromMsPedidos()`, como pode ser visto no último `import` da classe, é um método presente na classe `Pedido`, e esse será o método fábrica para a criação de pedidos. Ele tem como objetivo construir um objeto a partir do seu equivalente vindo do tópico, no formato Avro. Basicamente, ele instancia o `Pedido` usando o construtor privado e instancia seus atributos por meio de métodos de mesmo nome presentes nas classes `Endereco` e `ItemDoPedido`. Seu código está na sequência:

```
public static Pedido
fromMsPedidos(br.com.alura.mspedidos.infra.kafka.avro.P
edido pedidoAvro) {
    Pedido pedido = new Pedido();
    pedido.endereco =
Endereco.fromMsPedidos(pedidoAvro.getEndereco());
    pedido.itensDoPedido =
ItemDoPedido.fromMsPedidos(pedidoAvro.getItensDoPedido(
));
    return pedido;
}
```

O método `Endereco.fromMsPedidos()` encontra-se a seguir:

```
public static Endereco
fromMsPedidos(br.com.alura.mspedidos.infra.kafka.avro.E
ndereco enderecoAvro) {
    var endereco = new Endereco();
    endereco.rua = valueOf(enderecoAvro.getRua());
    endereco.numero = enderecoAvro.getNumero();
    endereco.complemento =
valueOf(enderecoAvro.getComplemento());
    endereco.bairro =
valueOf(enderecoAvro.getBairro());
    endereco.cidade =
valueOf(enderecoAvro.getCidade());
    endereco.estado =
valueOf(enderecoAvro.getEstado());
    endereco.cep = valueOf(enderecoAvro.getCep());
    return endereco;
}
```

E o método `ItemDoPedido.fromMsPedidos()` está aqui:

```
public static Set<ItemDoPedido> fromMsPedidos(

List<br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedi
do> itensDoPedidoAvro) {
    return itensDoPedidoAvro.stream()
            .map(ItemDoPedido::toItemDoPedido)
            .collect(Collectors.toSet());
}

private static ItemDoPedido toItemDoPedido(

br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedido
itemDoPedidoAvro){
    ItemDoPedido itemDoPedido = new ItemDoPedido();
    itemDoPedido.descricao =
valueOf(itemDoPedidoAvro.getItemCardapio().getDescricao
());
    itemDoPedido.nome =
valueOf(itemDoPedidoAvro.getItemCardapio().getNome());
    itemDoPedido.numero =
itemDoPedidoAvro.getItemCardapio().getNumero();
    itemDoPedido.quantidade =
itemDoPedidoAvro.getQuantidade();
    return itemDoPedido;
}
```

Perceba que o `ItemDoPedido` presente no *ms-entregadores* não contém um `ItemCardapio`. Em vez disso, os atributos presentes em `ItemCardapio` são atributos de `ItemDoPedido`. Dessa forma, simplifica-se a implementação desse microsserviço, visto que, para ele, esse objeto não é necessário.

Assim como temos uma classe `KafkaObjects` no *ms-pedidos* que fabrica os objetos que o Spring precisa para lidar com o Kafka, precisamos de classes para lidar com consumo de mensagens do Kafka, utilizando o Schema Registry para validá-las. Então vamos criar uma classe de mesmo nome no *ms-pedidos*.

Conforme vimos na construção do consumidor de mensagens existente em *ms-pedidos*, precisamos:

- do mapa de propriedades do Kafka e do Schema Registry;
- do objeto `KafkaListenerContainerFactory`, que utiliza o mapa já mencionado e é referenciado nominalmente na anotação `@KafkaListener` presente no `Receptor`.

A construção do objeto `KafkaListenerContainerFactory` é praticamente idêntica ao já feito no *ms-pedidos*. Só precisamos alterar os tipos dos objetos que estamos recebendo na chave e no valor da mensagem para, respectivamente, `String` e `Pedido`.

Temos duas classes `Pedido`:

- `br.com.alura.msentregadores.dominio.Pedido`;
- `br.com.alura.mspedidos.infra.kafka.avro.Pedido`, gerada via `avro-maven-plugin`.

Como esta última representa a mensagem enviada, devemos usá-la:

```
import br.com.alura.mspedidos.infra.kafka.avro.Pedido;

// trechos omitidos por brevidade

@Bean
KafkaListenerContainerFactory<ConcurrentMessageListener
Container<String, Pedido>> kafkaListener() {
    ConcurrentKafkaListenerContainerFactory<String,
Pedido> factory =
            new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(
            consumerConfigs()));
    return factory;
}
```

Já o método `consumerConfigs()` vai recorrer a desserializadores apropriados para as mensagens a serem recebidas:

- no *ms-pedidos*, usamos o `StringSerializer` para serializar a chave da mensagem. Então, de forma análoga, usaremos o `StringDeserializer` para desserializá-la;
- seguindo a mesma ideia, para o valor da mensagem, usaremos o `KafkaAvroDeserializer`.

Neste caso, apenas para mostrar que é possível, adotaremos também o método `Map.of()` para a construção do mapa de atributos, deixando o método como adiante. Dessa forma, facilitamos a criação desse mapa:

```
@Value("${spring.kafka.bootstrap-servers}")
private String kafkaServerUrl;

@Value("${spring.kafka.consumer.group-id}")
private String grupo;

@Value("${spring.kafka.consumer.auto-offset-reset}")
private String autoOffsetReset;

@Value("${spring.kafka.schema.registry.url}")
private String schemaRegistryUrl;

// trechos de código omitidos por brevidade
```

```
Map<String, Object> consumerConfigs() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
            ,
KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG
, true
    );
}
```

Precisamos também criar o `application.yml` para informar os valores das propriedades descritas no `KafkaObjects`:

```
spring:
  application.name: ms-entregadores

  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: ms-entregadores
```

```
      auto-offset-reset: earliest
    schema.registry.url: http://localhost:8081

topico.pedidos: pedidos
```

Para testar nosso `Receptor` de pedidos, podemos também aproveitar das mesmas ideias utilizadas no teste do receptor do *ms-pedidos*:

- criamos uma classe `KafkaObjectsParaTestes`, contendo um produtor simples de mensagens, usando os serializadores de mensagens utilizados no *ms-pedidos*, como segue:

```
@Configuration
class KafkaObjectsParaTestes {

    @Value("${spring.kafka.bootstrap-servers}")
    private String kafkaServerUrl;

    @Value("${spring.kafka.schema.registry.url}")
    private String schemaRegistryUrl;

    @Value("${topico.pedidos}")
    private String topicoDePedidos;

    @Value("${topico.entregadores}")
    private String topicoDeEntregadores;

    @Value("${spring.kafka.consumer.group-id}")
    private String grupo;

    @Value("${spring.kafka.consumer.auto-offset-
reset}")
```

```
    private String autoOffsetReset;

    @Bean
    KafkaTemplate<String, Pedido>
kafkaTemplateParaTestes(){
        return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(
                produtorConfigs()
        ));
    }

    private Map<String, Object> produtorConfigs() {
        Map<String, Object> props = new HashMap<>();

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);


props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

        return props;
    }
```

- também criamos a classe `ReceptorTest`, como na sequência:

```
@SpringBootTest
@EmbeddedKafka
@ActiveProfiles("test")
class ReceptorTest {

    @Autowired
    private KafkaTemplate<String,
br.com.alura.mspedidos.infra.kafka.avro.Pedido>
kafkaTemplateParaTestes;

    @MockitoSpyBean
    private Receptor receptor;

    @MockitoBean
    private SelecaoEntregadoresService
selecaoEntregadores;

    @Captor
    private ArgumentCaptor<String> chavePedidoCaptor;

    @Captor
    private
ArgumentCaptor<br.com.alura.mspedidos.infra.kafka.avro.
Pedido> pedidoCaptor;

    @Value("${topico.pedidos}")
    private String topico;

    @Test
    void receberUmNovoPedido() {
        var chavePedido = "67c4b4ba4606377f15d244d3";
        var umPedidoAvro = umPedidoAvro();
```

```
        kafkaTemplateParaTestes.send(topico,
chavePedido, umPedidoAvro);

        verify(receptor, timeout(10000).times(1))
                .receber(chavePedidoCaptor.capture(),
pedidoCaptor.capture());

        var pedido = fromMsPedidos(umPedidoAvro);

verify(selecaoEntregadores).selecionarEntregadorPara(pe
dido);
    }
}
```

Podemos, inclusive, rodar este teste:

```
mvn -f ms-entregadores/pom.xml test -Dtest=ReceptorTest

# trechos omitidos por brevidade

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped:
0, Time elapsed: 14.14 s -- in
br.com.alura.msentregadores.infra.kafka.ReceptorTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
------------------------
```

6.4 O Avro IDL (Interface Definition Language)

Conforme visto no início deste capítulo, há dois formatos de definição de *schemas* para Avro. Veremos agora o formato AVDL ou *Avro Interface Definition Language*.

Esse formato caracteriza-se por utilizar uma linguagem de definição mais legível e parecida com uma linguagem de programação, como Java ou C.

> Enquanto o Avro é um formato de serialização de dados, o AVSC e o AVDL são formatos de definição de *schemas*, e descrevem estruturas de dados que serão serializadas e desserializadas usando o Avro. Além disso, o AVSC é escrito em JSON e, portanto, pode ser considerado verboso, o que não acontece com o AVDL.

Tomemos como exemplo o *schema* adiante. Ele representa um endereço e é bem semelhante ao encontrado no *ms-pedidos*:

```
{
  "namespace":
"br.com.alura.msentregadores.infra.kafka.avro",
  "type": "record",
  "doc": "O endereço onde o pedido deve ser entregue",
  "name": "Endereco",
```

```
    "fields": [
      {
        "name": "rua",
        "type": "string"
      },
      {
        "name": "numero",
        "type": "int"
      },
      {
        "name": "bairro",
        "type": "string"
      },
      {
        "name": "cidade",
        "type": "string"
      },
      {
        "name": "estado",
        "type": "string"
      },
      {
        "name": "cep",
        "type": "string"
      },
      {
        "name": "complemento",
        "type": ["null", "string"],
        "default": null
      }
    ]
}
```

Para escrevermos o mesmo *schema*, mas usando o formato AVDL, usamos o código a seguir:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol EnderecoProtocol {

    record Endereco {
        string rua;
        int numero;
        union { null, string } complemento = null;
        string bairro;
        string cep;
        string cidade;
        string estado;
    }
}
```

Note que foi necessário muito menos código, além de representar de forma mais sucinta o *schema* de um endereço, facilitando a edição e manutenção de *schemas*.

IDEs como IntelliJ possuem recursos via *plugins* que oferecem suporte muito semelhante ao já oferecido na escrita de arquivos Java, como verificação de erros de sintaxe na escrita e *autocomplete* de código. Essa característica também pode ser considerada uma vantagem na utilização desse formato de *schema*.

Mais informações podem ser obtidas na documentação do formato, disponível em https://avro.apache.org/docs/1.12.0/idl-language/

6.5 Geração de stubs Avro a partir de schemas AVDL

Assim como escrevemos os arquivos AVSC, na pasta `src/main/avro`, vamos fazer o mesmo com o `Endereco.avdl`, já descrito. Neste capítulo, vimos que, além da classe `Endereco` como domínio, temos outras classes:

- `ItemDoPedido`
- `Pedido`
- `Entregador`

De forma semelhante em *ms-pedidos*, onde criamos um arquivo AVSC para cada objeto, vamos também criar um arquivo AVDL para cada um desses objetos. O `ItemDoPedido` possui um arquivo AVDL ainda mais simples que o `Endereco`, visto que possui menos campos e todos são obrigatórios. E o nome do arquivo é `ItemDoPedido.avdl`:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol ItemDoPedidoProtocol {

    record ItemDoPedido {
        string nome;
        string descricao;
        long numero;
        int quantidade;
```

```
    }
}
```

Tanto o `Endereco.avdl` quanto o `ItemDoPedido.avdl` contêm apenas tipos primitivos como atributos. Mas podemos também declarar tipos criados pelo programador. Veja por exemplo o `Pedido.avdl`. Nesse caso, há a declaração de um `array` de itens de pedido:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol PedidoProtocol {

    import idl "Endereco.avdl";
    import idl "ItemDoPedido.avdl";

    record Pedido {
        array<ItemDoPedido> itensDoPedido;
        Endereco endereco;
    }
}
```

Por último, precisamos escrever o `Entregador.avdl`. Ele contém apenas o nome do entregador e o pedido que ele deve entregar:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol EntregadorProtocol {

    import idl "Pedido.avdl";
```

```
    record Entregador {
        string nome;
        Pedido pedido;
    }
}
```

> É possível também inserir todos os `records` em um único arquivo. Nesse caso, seria usado um único `protocol`.

O avro-maven-plugin na geração de stubs a partir de *schemas* AVDL

É possível usar o `avro-maven-plugin` na produção de *stubs* Java a partir de *schemas* AVDL. Para isso, precisamos criar mais uma *execution* no *plugin*. Tomando como base a *execution* já criado para a conversão do `Pedido.avsc` para *stubs*:

- declaramos o `goal` como `idl-protocol`, em vez de `schema`;
- identificamos cada *execution* com um `id` diferente.

O código completo da configuração do *plugin* encontra-se a seguir:

```
<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
```

```
    <version>${avro.version}</version>
    <executions>
        <execution>
            <id>avsc-to-java</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
            </goals>
            <configuration>
                <createSetters>false</createSetters>
                <imports>

<import>${project.basedir}/src/main/avro/Pedido.avsc</i
mport>
                </imports>
            </configuration>
        </execution>
        <execution>
            <id>avdl-to-java</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>idl-protocol</goal>
            </goals>
            <configuration>
                <createSetters>false</createSetters>
                <imports>

<import>${project.basedir}/src/main/avro/Endereco.avdl<
/import>

<import>${project.basedir}/src/main/avro/ItemDoPedido.a
vdl</import>

<import>${project.basedir}/src/main/avro/Pedido.avdl</i
mport>
```

```
<import>${project.basedir}/src/main/avro/Entregador.avdl</import>
                </imports>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Com os *schemas* AVDL escritos e o *plugin* devidamente configurado, podemos executar a compilação do projeto para que os *stubs* sejam gerados. Note que nos *logs* de saída é possível verificar o momento em que ocorre a geração dos *stubs* a partir dos *schemas* `avsc` e `avdl`:

```
mvn -f ms-entregadores/pom.xml clean compile

# trechos omitidos por brevidade
[INFO] --- avro:1.12.0:schema (avsc-to-java) @ ms-entregadores ---
[INFO] Importing File: /Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livro/branch/rodrigo-vieira-pinto/codigo-fonte/ms-entregadores/src/main/avro/Pedido.avsc
[INFO]
[INFO] --- avro:1.12.0:idl-protocol (avdl-to-java) @ ms-entregadores ---
[INFO] Importing File: /Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livro/branch/rodrigo-vieira-pinto/codigo-fonte/ms-entregadores/src/main/avro/Endereco.avdl
[INFO] Importing File: /Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livro/branch/rodrigo-vieira-pinto/codigo-fonte/ms-
```

```
entregadores/src/main/avro/ItemDoPedido.avdl
[INFO] Importing File:
/Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livr
o/branch/rodrigo-vieira-pinto/codigo-fonte/ms-
entregadores/src/main/avro/Pedido.avdl
[INFO] Importing File:
/Users/rodrigovieirapinto/BACKUP/Rodrigo/workspace/livr
o/branch/rodrigo-vieira-pinto/codigo-fonte/ms-
entregadores/src/main/avro/Entregador.avdl
[INFO]
[INFO] --- resources:3.3.1:resources (default-
resources) @ ms-entregadores ---
[INFO] Copying 1 resource from src/main/resources to
target/classes
[INFO] Copying 0 resource from src/main/resources to
target/classes
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @
ms-entregadores ---
[INFO] Recompiling the module because of changed source
code.
[INFO] Compiling 26 source files with javac [debug
parameters release 21] to target/classes
[INFO] ------------------------------------------------
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------
------------------------
```

6.6 Cadastro do schema no Schema Registry e o avro-tools

Com os *schemas* escritos, podemos cadastrar o *subject* `entregadores-value` no Schema Registry. Entretanto, temos um problema: o Schema Registry não suporta o cadastro de *schemas* no formato AVDL. Logo, é necessário:

- converter os *schemas* para o formato AVSC, gerando um único *schema*. Assim usamos um *schema* por tópico, conforme definimos no capítulo 5;
- cadastrar o *schema* resultante no Schema Registry.

Para o primeiro passo, usaremos a ferramenta ***avro-tools***. É uma ferramenta de linha de comando oficial da Apache Avro, usada para manipular arquivos Avro e *schemas*. Com ela, é possível realizar tarefas como:

- converter arquivos entre formatos JSON e binário;
- validar e visualizar *schemas*;
- gerar código-fonte a partir de *schemas* (.avsc) e IDLs (.avdl);
- compactar/descompactar arquivos Avro.

É amplamente utilizada em *pipelines* de dados e no desenvolvimento com Kafka e Schema Registry. O *avro-tools* é distribuído como um JAR executável

standalone, tornando-o fácil de usar em qualquer ambiente com Java instalado.

Usaremos os comandos a seguir para baixá-lo:

```
AVRO_TOOLS="avro-tools-1.12.0.jar"
URL="https://repo1.maven.org/maven2/org/apache/avro/avro-tools/1.12.0/avro-tools-1.12.0.jar"
wget -O "$AVRO_TOOLS" "$URL"
```

Em seguida, usamos o *avro-tools* para gerar um arquivo AVSC a partir de um AVDL. Será produzido um arquivo AVSC para cada arquivo AVDL, na pasta `out`. Entretanto, esses arquivos serão emitidos de forma "desnormalizada" (o formato esperado pelo Schema Registry). Como precisamos enviar um entregador para o tópico `entregadores`, precisamos do arquivo `Entregador.avsc` gerado.

Para gerar esses arquivos, usamos o comando:

```
java -jar avro-tools-1.12.0.jar idl2schemata ../ms-entregadores/src/main/avro/Entregador.avdl out
```

A seguir, é possível vermos como o `Entregador.avsc` é criado:

```
{
  "type" : "record",
  "name" : "Entregador",
  "namespace" :
"br.com.alura.msentregadores.infra.kafka.avro",
  "fields" : [ {
```

```
  "name" : "nome",
  "type" : "string"
}, {
  "name" : "pedido",
  "type" : {
    "type" : "record",
    "name" : "Pedido",
    "fields" : [ {
      "name" : "itensDoPedido",
      "type" : {
        "type" : "array",
        "items" : {
          "type" : "record",
          "name" : "ItemDoPedido",
          "fields" : [ {
            "name" : "nome",
            "type" : "string"
          }, {
            "name" : "descricao",
            "type" : "string"
          }, {
            "name" : "numero",
            "type" : "long"
          }, {
            "name" : "quantidade",
            "type" : "int"
          } ]
        }
      }
    }, {
      "name" : "endereco",
      "type" : {
        "type" : "record",
        "name" : "Endereco",
        "fields" : [ {
          "name" : "rua",
```

```
            "type" : "string"
          }, {
            "name" : "numero",
            "type" : "int"
          }, {
            "name" : "complemento",
            "type" : [ "null", "string" ],
            "default" : null
          }, {
            "name" : "bairro",
            "type" : "string"
          }, {
            "name" : "cep",
            "type" : "string"
          }, {
            "name" : "cidade",
            "type" : "string"
          }, {
            "name" : "estado",
            "type" : "string"
          } ]
        }
      } ]
    }
  } ]
}
```

Note que o arquivo gerado não referencia outros arquivos, mas contém todos os objetos nele mesmo.

Outros exemplos de uso do *avro-tools* podem ser vistos em https://avro.apache.org

Agora basta utilizarmos o Avro Schema Merger para gerarmos o *schema* pronto para ser cadastrado via REST no Schema Registry:

```
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../schemas/out true Entregador.avsc >
../schemas/entregadores-value.js

cd ../schemas

curl -X POST
http://localhost:8081/subjects/entregadores-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@entregadores-value.js
```

Felizmente, todos esses comandos estão reunidos no arquivo `entregadores-value-cadastrar.sh`, que se localiza com todos os demais scripts *bash* de cadastro de *subjects*. Seu conteúdo completo é identificado adiante:

```
AVRO_TOOLS="avro-tools-1.12.0.jar"
URL="https://repo1.maven.org/maven2/org/apache/avro/avr
o-tools/1.12.0/avro-tools-1.12.0.jar"

if [ ! -f "$AVRO_TOOLS" ]; then
  echo "Avro tools não encontrado, iniciando o
download..."
  wget -O "$AVRO_TOOLS" "$URL"
else
  echo "Avro tools já existe. Realizando a conversão de
avdl em avsc.."
```

```
fi

java -jar "$AVRO_TOOLS" idl2schemata ../ms-
entregadores/src/main/avro/Entregador.avdl out

cd ../avroschemamerger
./instalar_avroschemamerger.sh
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../schemas/out true Entregador.avsc >
../schemas/entregadores-value.js

cd ../schemas

curl -X POST
http://localhost:8081/subjects/entregadores-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@entregadores-value.js

 rm -rf ../schemas/out
```

6.7 Implementação do produtor de mensagens

Agora que o *schema* foi devidamente cadastrado no Schema Registry, podemos finalmente escrever a classe que implementa a interface `Despachante`. Vamos chamá-la de `DespachanteKafka`.

Embora usemos *schemas* no formato AVDL em vez do AVSC, o `DespachanteKafka` e todos os *beans*

necessários ao seu funcionamento não são muito diferentes dos já implementados no `PedidoSenderKafka`, existente no *ms-pedidos*. Neste caso, teremos algumas diferenças:

- não enviaremos uma chave para as mensagens. Logo, podemos usar a classe `VoidSerializer` como serializador de chave, e a classe `Void` como chave da mensagem.
- vamos trabalhar com objetos da classe `br.com.alura.msentregadores.infra.kafka.avro.Entregador` como valores.

Para construção dos *beans* usa-se o seguinte código em `KafkaObjects`:

```
@Bean
KafkaTemplate<Void, Entregador> kafkaTemplate(){
    return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(produtorConfigs()));
}

private Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
VoidSerializer.class);
```

```
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);


props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

    return props;
}
```

Em seguida, a classe `DespachanteKafka`:

```
package br.com.alura.msentregadores.infra.kafka;

import br.com.alura.msentregadores.dominio.Despachante;
import br.com.alura.msentregadores.dominio.Entregador;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class DespachanteKafka implements Despachante {

    private KafkaTemplate<Void,
br.com.alura.msentregadores.infra.kafka.avro.Entregador
> entregadorKafkaTemplate;
    private String topico;

    public DespachanteKafka(KafkaTemplate<Void,
```

```
br.com.alura.msentregadores.infra.kafka.avro.Entregador
> entregadorKafkaTemplate,

@Value("${topico.entregadores}") String topico){
        this.entregadorKafkaTemplate =
entregadorKafkaTemplate;
        this.topico = topico;
    }

    @Override
    public void despachar(Entregador entregador) {
        entregadorKafkaTemplate.send(topico, null,
entregador.toAvro());
    }
}
```

> Os testes presentes em `DespachanteKafkaTest` são semelhantes aos da classe `PedidosSenderKafkaTest`. Vale a pena verificar a *branch* `cap-6` para mais informações.

Assim como fizemos no *ms-pedidos*, onde convertemos o objeto `Pedido` de negócio para um de mesmo nome que se refere ao *stub* criado, fazemos o mesmo com o `Entregador`.

O método `toAvro()` presente nessa classe encontra-se a seguir:

```
public
br.com.alura.msentregadores.infra.kafka.avro.Entregador
toAvro() {
```

```
    return
br.com.alura.msentregadores.infra.kafka.avro.Entregador
.newBuilder()
        .setNome(nome)
        .setPedido(status.lerPedido().toAvro())
        .build();
}
```

6.8 Resumo

Após a leitura deste capítulo, é possível concluir que, no geral (do ponto de vista de produtores e consumidores de mensagens):

- as diferenças entre lidar com *schemas* JSON e AVRO são os serializadores/desserializadores a serem configurados e as ferramentas para serialização/desserialização das mensagens (baseadas em anotações no caso do JSON ou de *stubs* no caso do Avro). O restante do código tende a ser muito semelhante.
- no caso do Avro, uma vez que os *stubs* são gerados, todo o restante do código é praticamente o mesmo, independente do formato de *schema*.

Neste capítulo, vimos como criar consumidores de mensagens usando AVRO e *schemas* no formato

AVSC. Também vimos a criação de produtores de mensagens usando *schemas* no formato AVDL.

No próximo capítulo, veremos características mais avançadas do Schema Registry.

CAPÍTULO 7

Avançando com Schema Registry: Modos de compatibilidade, tipos lógicos do Avro e customizações

Ao longo da construção do Alura Eats, tivemos contato com diversas funcionalidades do Schema Registry. Vimos como cadastrar, selecionar ou excluir *subjects*, *schemas* e trabalhamos com as estruturas JSON e Avro (usando AVSC e AVDL como formatos de *schema*).

Neste capítulo, veremos algumas características avançadas e curiosidades sobre essa ferramenta.

> Os códigos presentes neste capítulo estão na *branch* `cap-7` do projeto no GitHub.

7.1 Idempotência das APIs do Schema Registry

Conforme vimos no capítulo 2, podemos usar a API REST do Schema Registry para realizar manipulações de *subjects* e *schemas* com facilidade. Dentre essas

manipulações, podemos cadastrar novos *subjects* usando comandos como o seguinte:

```
curl -X POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@restaurantes.cardapio.itens-value.js
```

> Esse comando está no *script* `restaurantes.cardapio.itens-value-cadastrar.sh`, presente na pasta `schemas`.

Entretanto, podemos acidentalmente executar esse comando mais de uma vez. Em uma situação dessas, o que pode acontecer?

- O Schema Registry retornar algum erro, informando que o *schema* já foi cadastrado?
- O Schema Registry cadastrar *n* vezes o mesmo *schema*, onde *n* é a quantidade de vezes que se tentou cadastrar o mesmo *schema*?
- O Schema Registry retornará *schemas IDs* de *schema* diferentes para cada tentativa?

Felizmente, o Schema Registry trabalha com o conceito de idempotência. Isso significa que:

- não serão retornados erros;

- não teremos diversos cadastros do mesmo *schema*, uma vez que ele já existe na aplicação;
- o Schema Registry sempre retornará o mesmo *Schema ID*, independente da quantidade de tentativas.

Podemos, portanto, rodar o *script* citado diversas vezes. Ele sempre retornará o mesmo *Schema ID* e não realizará alterações no *schema*:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo codigo-fonte
% cd schemas
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@restaurantes.cardapio.itens-value.js

{"id":2}%

rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@restaurantes.cardapio.itens-value.js

{"id":2}%
```

Essa mesma ideia se aplica às demais APIs:

- se tentarmos apagar um *schema*, novas tentativas de apagá-lo não terão efeito adicional;
- recuperação de um *schema* de diferentes formas. Por serem operações de busca, são naturalmente idempotentes.

> A idempotência, em resumo, garante que operações repetidas tenham o mesmo efeito, evitando registros duplicados e permitindo chamadas seguras, sem medo de impactos inesperados.
>
> Para mais informações sobre esse conceito, acesse https://restfulapi.net/idempotent-rest-apis/

7.2 Enviando mensagens sem aval do Schema Registry

Nos capítulos anteriores, também vimos a importância de os produtores estarem preparados para trabalhar com o Schema Registry. Para isso, inserimos algumas configurações nos mapas dos produtores:

- `AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG`;

- os serializadores adequados para o protocolo escolhido (`KafkaJsonSchemaSerializer` no caso de mensagens JSON e `KafkaAvroSerializer` no caso de Avro).

Além disso, cadastramos os *schemas* para cada um dos tópicos, seguindo inclusive a nomenclatura padrão para nomes de *subjects*:

- se o tópico se chama `restaurantes.cardapio.itens` e precisamos cadastrar um *subject* para a chave da mensagem, seu nome deve ser `restaurantes.cardapio.itens-key` ;
- se precisamos cadastrar um *subject* para o mesmo tópico, mas para o valor da mensagem, seu nome deve ser `restaurantes.cardapio.itens-value` .

Entretanto, é possível enviar mensagens para o Kafka sem que o Schema Registry realize sua validação. Basta que não façamos nenhuma referência ao Schema Registry. A mensagem chegará normalmente ao tópico!

Como exemplo, consideremos o *ms-restaurantes*. O mapa de configurações do produtor encontra-se da seguinte forma:

```
public Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();
```

```
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaJsonSchemaSerializer.class);
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, false);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

props.put(KafkaJsonSchemaSerializerConfig.FAIL_INVALID_
SCHEMA, true);

    return props;
}
```

Se, no entanto, a configuração for feita da seguinte forma:

```
public Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();


props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
```

```
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

    return props;
}
```

E caso realizemos o cadastro de um novo item de cardápio com campos faltando (no caso, a descrição do item):

```
curl -X PUT http://localhost:8080/itens-cardapio \
--header 'Content-Type: application/json' \
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa"
}'
```

A mensagem chegará ao tópico Kafka sem problemas:

Figura 7.1: Mensagem chegando ao Kafka, sem validação do Schema Registry.

> Embora o Schema Registry seja uma ferramenta que realize a efetiva validação das mensagens, ele precisa estar devidamente configurado em todos os produtores de mensagens para que isso aconteça. O simples fato de haver *schemas* cadastrados não garante essa validação.

7.3 Tipos lógicos no Avro

Vimos no capítulo 5 que o Avro possui diversos tipos primitivos. Podemos representar valores numéricos

com campos `int` ou mesmo nomes de pessoas com campos `string`.

Entretanto, assim como ocorre em diversas linguagens de programação, os tipos primitivos podem não ser suficientes para representar corretamente alguma informação. Se, por exemplo, quisermos representar uma data na linguagem Java, normalmente usamos um objeto da classe `java.time.LocalDate`, ou mesmo um objeto da classe `java.util.UUID` para representar um identificador único universal.

É aí que entram os tipos lógicos do Avro. Eles nos ajudam a representar, utilizando tipos primitivos como base, dados mais complexos, que precisam de um significado mais específico.

Tomemos como exemplo a representação de uma data. Se precisarmos representar a data de nascimento de uma pessoa em um campo chamado `dataNascimento`, sendo esse um campo obrigatório, devemos representá-lo como segue:

```
{
  "name": "dataNascimento",
  "type": {
    "type": "int",
    "logicalType": "date"
  }
}
```

Observe que as primeiras linhas são semelhantes ao que já vimos nos capítulos anteriores: temos tanto o elemento `name` (que recebe o nome do atributo) e o `type`, representando o tipo. A partir daí, temos as seguintes mudanças:

- o primeiro `type` representa o tipo do atributo. E esse tipo pode ter mais detalhes;
- o segundo `type` representa o tipo físico real, usado para armazenar o valor (`int` nesse caso);
- o elemento `logicalType` informa que esse `int` armazena uma data. Ou seja, ele armazenará o número de dias desde 1 de janeiro de 1970.

Veremos ainda neste capítulo outro tipo lógico: o `decimal`.

> Para mais informações sobre tipos lógicos, o *link* de referência para a especificação mais recente é https://avro.apache.org/docs/1.12.0/specification/_print/#:~:text=Logical%20Types

7.4 Modos de compatibilidade entre schemas de mesmo subject

Conforme vimos no capítulo 3, podemos realizar diversas operações no Schema Registry usando os

endpoints disponíveis pela ferramenta. Uma dessas operações é a inclusão de novos *schemas* para um mesmo *subject*. Dessa forma, garantimos a evolução das mensagens a serem trafegadas via tópicos.

A evolução desses *schemas* precisa ser feita com cuidado, garantindo que os produtores e consumidores continuem conseguindo se comunicar entre si. É aí que entra o conceito de compatibilidade.

Compatibilidade é a regra que define se um novo *schema* pode ser registrado como uma nova versão do *schema* anterior, levando em consideração se ele é capaz de ler dados antigos, escrever dados legíveis por versões antigas ou fazer ambas as coisas.

O Schema Registry valida automaticamente se o novo *schema* é compatível com o anterior antes de registrá-lo. Para isso, ele verifica a política de compatibilidade cadastrada para o *subject* correspondente.

- Para ler dados antigos: `BACKWARD`.
- Para escrever dados legíveis por versões antigas: `FORWARD`.
- Para ambos: `FULL`.

Imagine que um produtor comece a enviar mensagens com um novo campo obrigatório. Se o

consumidor ainda estiver usando um *schema* antigo, ele pode não saber como lidar com esse novo campo, resultando em erro de desserialização.

Ao configurar uma política de compatibilidade, evitamos esse tipo de problema, permitindo apenas evoluções seguras entre as versões de um *schema*.

Uma vez configurando corretamente a compatibilidade para cada *subject*, garantimos que:

- mensagens antigas continuam sendo compreendidas pelos consumidores atuais;
- mensagens novas continuam sendo compreendidas por consumidores antigos (dependendo da configuração);
- a comunicação entre sistemas não é quebrada por engano.

Caso não seja cadastrada nenhuma política de compatibilidade nos *schemas*, será usada a `BACKWARD`, política padrão do Schema Registry da Confluent. Caso o Schema Registry seja de outro fabricante, pode ser necessário verificar qual a política *default* usada.

As políticas `BACKWARD`, `FORWARD` e `FULL` são as mais básicas, mas existem outras:

- `NONE` : sem verificação de compatibilidade. Qualquer *schema* pode ser registrado.
- `BACKWARD_TRANSITIVE` : o novo *schema* pode ler dados de todas as versões anteriores (não só a última).
- `FORWARD_TRANSITIVE` : os dados escritos com o novo *schema* podem ser lidos por todas as versões anteriores.
- `FULL_TRANSITIVE` : combina `BACKWARD_TRANSITIVE` e `FORWARD_TRANSITIVE` . Compatível com todas as versões anteriores e futuras.

Uma discussão mais aprofundada desse tema pode ser encontrada em https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html

7.5 A inclusão do preço nos itens de cardápio

Até o momento, implementamos diversos requisitos, desde o cadastro de cardápios e seus itens até a entrega dos pedidos para os entregadores. No entanto, deixamos de lado uma informação essencial: o preço dos produtos. Nesta seção,

veremos como lidar com esse dado fundamental em todas as aplicações desenvolvidas até agora.

Inclusão do preço no ms-restaurantes

Para incluirmos a informação de preço no *ms-restaurantes*, precisamos inicialmente incluir esse campo na aplicação e no *schema* associado ao tópico `restaurantes.cardapio.itens-value`.

Na aplicação, temos as classes `ItemCardapioCriadoDTO` e `ItemCardapio`, que passam a ter também o campo `preco` do tipo `BigDecimal`.

A classe `ItemCardapioCriadoDTO` representa a mensagem que virá pelo *endpoint* já existente na aplicação e está apresentada na sequência, atualizada com o novo campo.

```
public class ItemCardapioCriadoDTO {

    @JsonProperty("id-restaurante")
    private Integer idRestaurante;

    @JsonProperty
    private Integer numero;

    @JsonProperty
    private String nome;

    @JsonProperty
    private String descricao;
```

```
    @JsonProperty
    private BigDecimal preco;

    public ItemCardapioCriado toModel() {
        return new ItemCardapioCriado(
                new IdentificacaoItem(idRestaurante,
numero),
                new ItemCardapio(nome, descricao,
preco.toPlainString()));
    }
}
```

É possível notar que o construtor de `ItemCardapio` passou a receber o preço também, mas como `String`. O *record* completo está a seguir:

```
public record ItemCardapio(
    @JsonProperty(required = true)
    String nome,

    @JsonProperty(required = true)
    String descricao,

    @JsonProperty(required = true)
    String preco
) {
}
```

O formato JSON Schema não suporta tipos diferentes de `integer`, `number` ou `string` para números. Logo, é necessário escolher qual deles usar para representar o preço de um item.

Podemos usar o tipo `number`, visto que ele aceita tanto números decimais quanto inteiros. Entretanto, ao converter uma informação numérica para `BigDecimal` podemos ter perda de precisão.

> Mais informações sobre perda de precisão envolvendo tipos numéricos que representam números decimais para `BigDecimal` podem ser vistas neste link: https://www.alura.com.br/artigos/arredondamento-no-java-do-double-ao-bigdecimal

Assim sendo, vamos usar o tipo `String` para representar o preço no código Java. E no *schema*, usaremos o tipo *string*, criando uma nova versão no Schema Registry.

Para criarmos essa nova versão, inicialmente criamos o arquivo `restaurantes.cardapio.itens-value-v2.js` na pasta de `schemas`, que naturalmente é muito parecido com o `restaurantes.cardapio.itens-value.js`, tendo apenas o campo de `preco` como diferença. Esse campo também é considerado obrigatório. Logo, precisamos incluí-lo no *array* `required`:

```
{
   "schemaType": "JSON",
   "schema": "{
      \"$schema\": \"http://json-schema.org/draft-
```

```
07/schema#\",
      \"title\": \"Item Cardapio\",
      \"type\": \"object\",
      \"additionalProperties\": false,
      \"properties\": {
         \"nome\": {
            \"type\": \"string\"
         },
         \"descricao\": {
            \"type\": \"string\"
         },
         \"preco\": {
            \"type\": \"string\"
         }
      },
      \"required\": [\"nome\", \"descricao\",
\"preco\"]
   }"
}
```

Todos os *schemas* que criamos são aplicados no Schema Registry por meio de *scripts bash*. E com o mostrado recentemente não será diferente.

Esse *script*, porém, não vai cadastrar um novo *schema*, mas criar uma nova versão de um *schema* existente. E, felizmente, o *endpoint* a ser usado é o mesmo que usamos para cadastrar novos *schemas*. Só precisamos referenciar o novo arquivo `restaurantes.cardapio.itens-value-v2.js`.

O *script bash* é o `restaurantes.cardapio.itens-value-cadastrar-v2.sh`, e encontra-se juntamente aos

demais, na pasta `schemas`:

```
echo "\nCadastrando a nova versão do schema"
curl -X POST
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@restaurantes.cardapio.itens-value-v2.js
```

Também precisamos inserir a chamada ao *script* acima no `infra-up.sh`. Assim, a infraestrutura da aplicação estará sempre atualizada. A seguir, o trecho de código contendo a alteração:

```
while true; do
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
    ./entregadores-value-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar-v2.sh
    break
  fi
  sleep 5
done
```

Entretanto, ao rodarmos o *script*, temos o seguinte problema:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./restaurantes.cardapio.itens-value-cadastrar-v2.sh
```

```
Cadastrando a nova versão do *schema*:
{"error_code":409,"message":"Schema being registered is
incompatible with an earlier schema for subject
\"restaurantes.cardapio.itens-value\", details:
[{errorType:\"REQUIRED_PROPERTY_ADDED_TO_UNOPEN_CONTENT
_MODEL\", description:\"The new schema has an unopen
content model and has a required property at path
'#/properties/preco' which is missing in the old
schema'}, {oldSchemaVersion: 1}, {oldSchema:
'{\"$schema\":\"http://json-schema.org/draft-
07/schema#\",\"title\":\"Item
Cardapio\",\"type\":\"object\",\"additionalProperties\"
:false,\"properties\":{\"nome\":
{\"type\":\"string\"},\"descricao\":
{\"type\":\"string\"}},\"required\":
[\"nome\",\"descricao\"]}'}, {validateFields: 'false',
compatibility: 'BACKWARD'}]"}%
```

Repare na mensagem de erro, destacada adiante. Ela informa que foi inserido um novo campo obrigatório, que não existe na versão anterior do *schema*:

```
{errorType:\"REQUIRED_PROPERTY_ADDED_TO_UNOPEN_CONTENT_
MODEL\", description:\"The new schema has an unopen
content model and has a required property at path
'#/properties/preco' which is missing in the old
schema'}
```

Esse tipo de alteração quebra a compatibilidade com mensagens anteriores, já que uma mensagem antiga não possui esse campo, não podendo, portanto, ser desserializada com o novo *schema*.

A título de curiosidade, podemos verificar o modo usado em `restaurantes.cardapio.itens-value` usando o seguinte comando:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
curl -X GET
http://localhost:8081/config/restaurantes.cardapio.iten
s-value
{"compatibilityLevel":"BACKWARD"}%
```

Como não inserimos um modo de compatibilidade diferente do padrão adotado pelo Schema Registry no momento do cadastro, o modo *default* foi usado.

Para resolvermos esse problema, temos algumas alternativas:

- tornar o campo `preco` não obrigatório e com um valor padrão, o que é ruim, visto que precisamos dessa informação para que nossos clientes possam ser cobrados;
- alterar provisoriamente a compatibilidade para um modo que permita a atualização do *schema*, atualizar o *schema* e depois retornar ao modo ideal.

O único modo que permite a atualização é o `NONE`, visto que, nesse modo, o Schema Registry não realiza verificações nas mensagens.

O modo `NONE` deve ser usado com cautela, principalmente em ambiente de produção, pois corre-se o risco de receber mensagens que causem falhas nos consumidores existentes enquanto ele estiver sendo usado.

Vamos então:

- atualizar o modo de compatibilidade de `restaurantes.cardapio.itens-value` para `NONE`;
- cadastrar a nova versão de `restaurantes.cardapio.itens-value`;
- retornar o modo de compatibilidade de `restaurantes.cardapio.itens-value` para `BACKWARD`.

Para isso, atualizamos o *script* `restaurantes.cardapio.itens-value-v2.sh` como a seguir:

```
echo "\nAlterando a compatibilidade do schema
restaurantes.cardapio.itens-value para NONE
temporariamente"
curl -X PUT
http://localhost:8081/config/restaurantes.cardapio.iten
s-value \
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "NONE"}'

echo "\nCadastrando a nova versão do schema"
curl -X POST
```

```
http://localhost:8081/subjects/restaurantes.cardapio.it
ens-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@restaurantes.cardapio.itens-value-v2.js

echo "\nRetornando a compatilidade do schema
restaurantes.cardapio.itens-value para BACKWARD"
curl -X PUT
http://localhost:8081/config/restaurantes.cardapio.iten
s-value \
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "BACKWARD"}'
```

Ao rodarmos o *script* novamente, a atualização ocorre normalmente:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./restaurantes.cardapio.itens-value-cadastrar-v2.sh

Alterando a compatibilidade do *schema*
restaurantes.cardapio.itens-value para NONE
temporariamente:
{"compatibility":"NONE"}
Cadastrando a nova versão do schema
{"id":5}
Retornando a compatilidade do schema
restaurantes.cardapio.itens-value para BACKWARD
{"compatibility":"BACKWARD"}%
```

Podemos, portanto, começar a enviar os itens de cardápio com o novo preço.

Subimos o *ms-restaurantes*:

```
mvn -f ms-restaurantes/pom.xml spring-boot:start
```

E enviamos outro item de cardápio, usando o Postman ou via linha de comando:

```
curl --X PUT 'http://localhost:8080/itens-cardapio' \
--header 'Content-Type: application/json' \
--data '{
    "id-restaurante": 1,
    "numero": 2,
    "nome": "X-Tudo Especial",
    "descricao": "Um lanche que mata a sua fome"
    "preco": 50.00
}'
```

Verificamos que o novo item de cardápio foi enviado para o Kafka com sucesso:

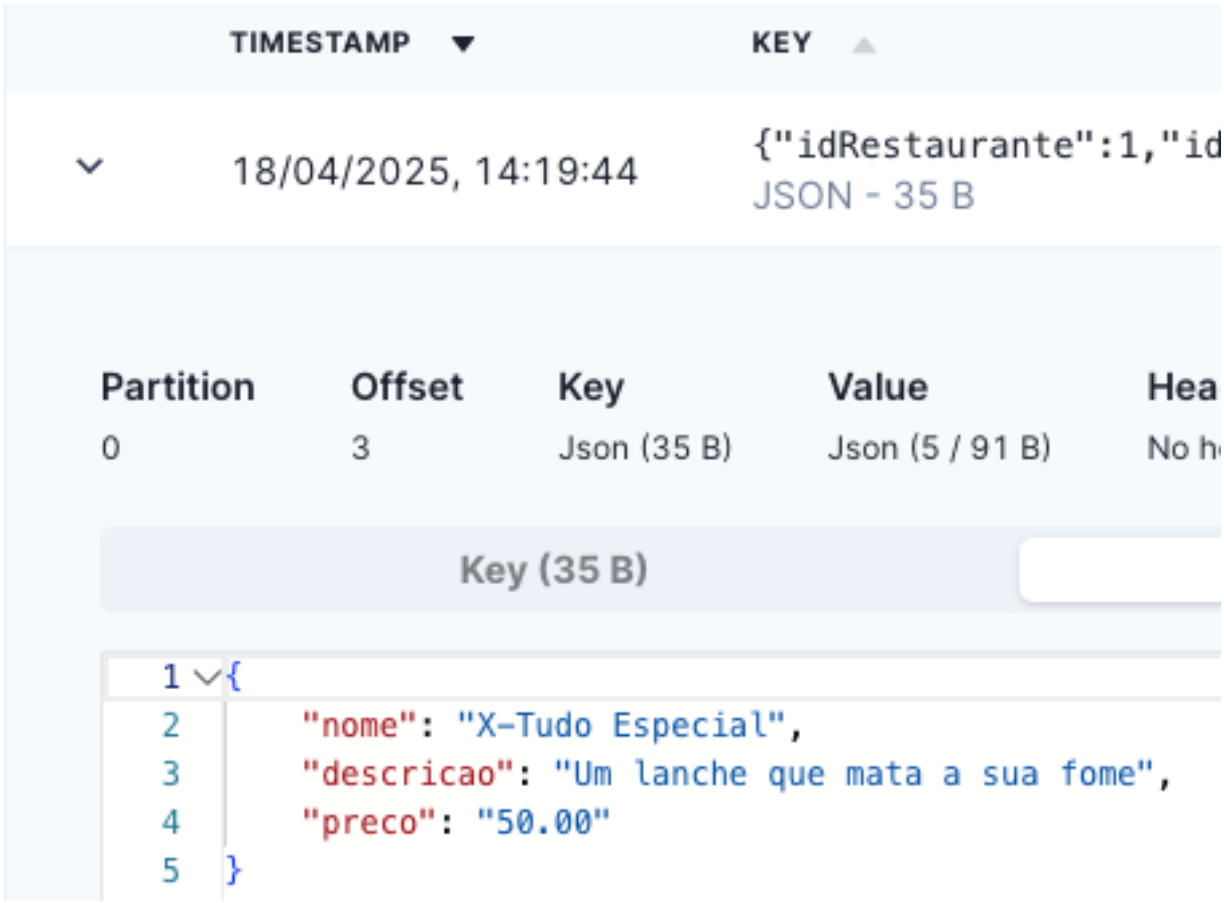


Figura 7.2: Mensagem chegando ao Kafka, sem validação do Schema Registry.

Mas e o *ms-pedidos*, que consome esse tópico? Como ele vai se comportar? Ao subirmos esta aplicação:

```
mvn -f ms-pedidos/pom.xml spring-boot:start
```

Ela deve ler o tópico e inserir o novo registro em banco de dados, o que de fato ocorre:

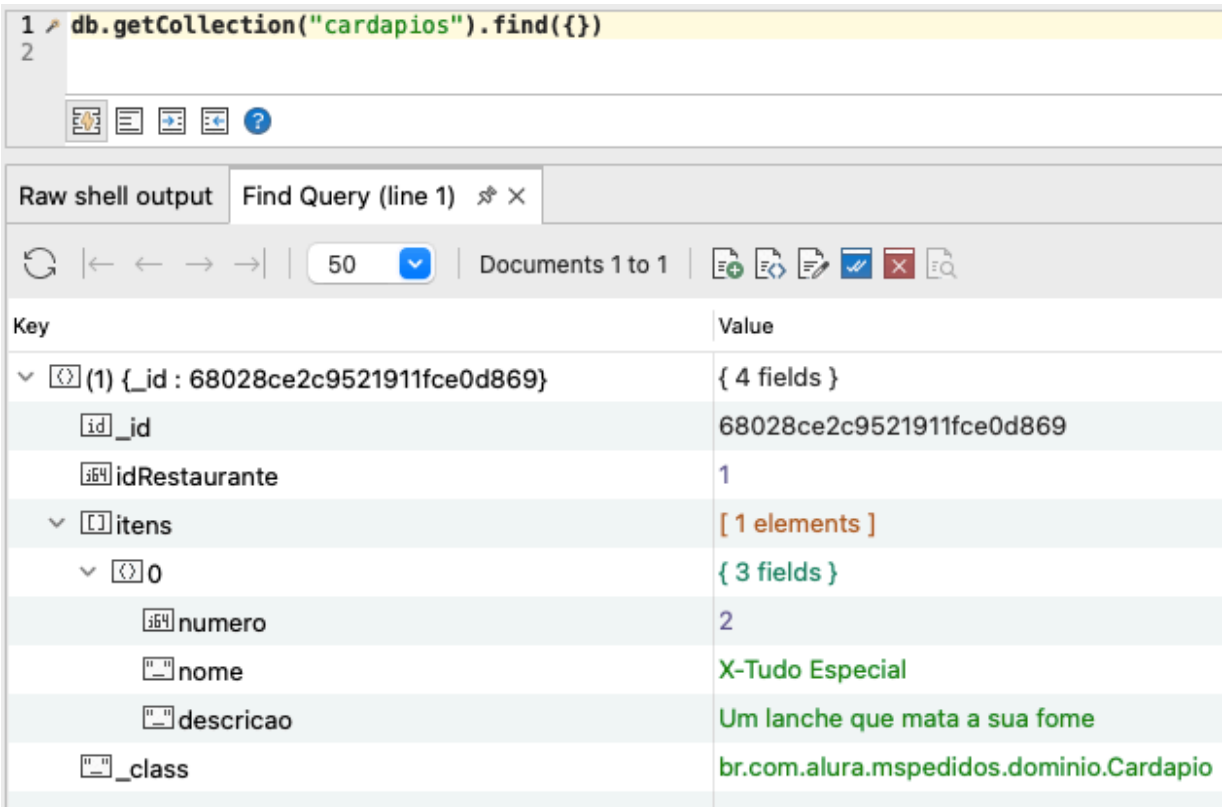


Figura 7.3: Novo item de cardápio no banco de dados.

Mas não temos a informação de preço, visto que o *ms-pedidos* está trabalhando ainda na versão anterior do *schema*. Vamos então ajustá-lo.

Inclusão do preço no recebimento de itens de cardápio no *ms-pedidos*

A alteração no consumo é bem semelhante ao que fizemos no *ms-restaurantes*: basta inserirmos o novo campo no *record* que representa o item de cardápio vindo do tópico, no caso, o `ItemCardapioRecebido`. Precisamos também enviar esse novo campo para a

classe `ItemCardapio`, cujo objeto é gerado pelo método `toItemCriado()`.

O *record* `ItemCardapio` com as alterações encontra-se a seguir:

```
public record ItemCardapioRecebido(
    @JsonProperty(required = true) String nome,
    @JsonProperty(required = true) String descricao,
    @JsonProperty(required = true) String preco) {

    public ItemCardapioCriado
toItemCriado(IdentificacaoItem id) {
        return new ItemCardapioCriado(id, new
ItemCardapio(id.idItem(), nome, descricao, new
BigDecimal(preco)));
    }
}
```

E o `ItemCardapio` está adiante. Note que a informação de `preco` é enviada também para o objeto `br.com.alura.mspedidos.infra.kafka.avro.ItemCardapio`, usado para envio de pedidos para o tópico de `pedidos`:

```
public record ItemCardapio(
        @JsonProperty Long numero,
        @JsonProperty String nome,
        @JsonProperty String descricao,
        @JsonProperty BigDecimal preco){

    @Override
```

```
    public boolean equals(Object obj){
        if(obj == null) return false;
        if(!(obj instanceof ItemCardapio)) return
false;
        ItemCardapio outro = (ItemCardapio) obj;
        return this.numero.equals(outro.numero)
                && this.nome.equals(outro.nome);
    }

    @Override
    public int hashCode(){
        return Objects.hash(numero, nome);
    }

    public
br.com.alura.mspedidos.infra.kafka.avro.ItemCardapio
toAvro() {
        return
br.com.alura.mspedidos.infra.kafka.avro.ItemCardapio.ne
wBuilder()
                .setNumero(numero)
                .setNome(nome)
                .setDescricao(descricao)
                .setPreco(preco)
                .build();
    }
}
```

Podemos então subir novamente o *ms-pedidos*:

```
mvn -f ms-pedidos/pom.xml spring-boot:start
```

E enviar um novo item de cardápio, como o seguinte:

```
curl --X PUT 'http://localhost:8080/itens-cardapio' \
--header 'Content-Type: application/json' \
```

```
--data '{
    "id-restaurante": 1,
    "numero": 1,
    "nome": "Feijoada Completa",
    "descricao": "Uma feijoada onde o céu é o limite",
    "preco": 50.00
}'
```

Com isso, estamos cadastrando um novo item para o cardápio já existente. Com o devido suporte ao novo campo de `preco`, ele é persistido em banco de dados:

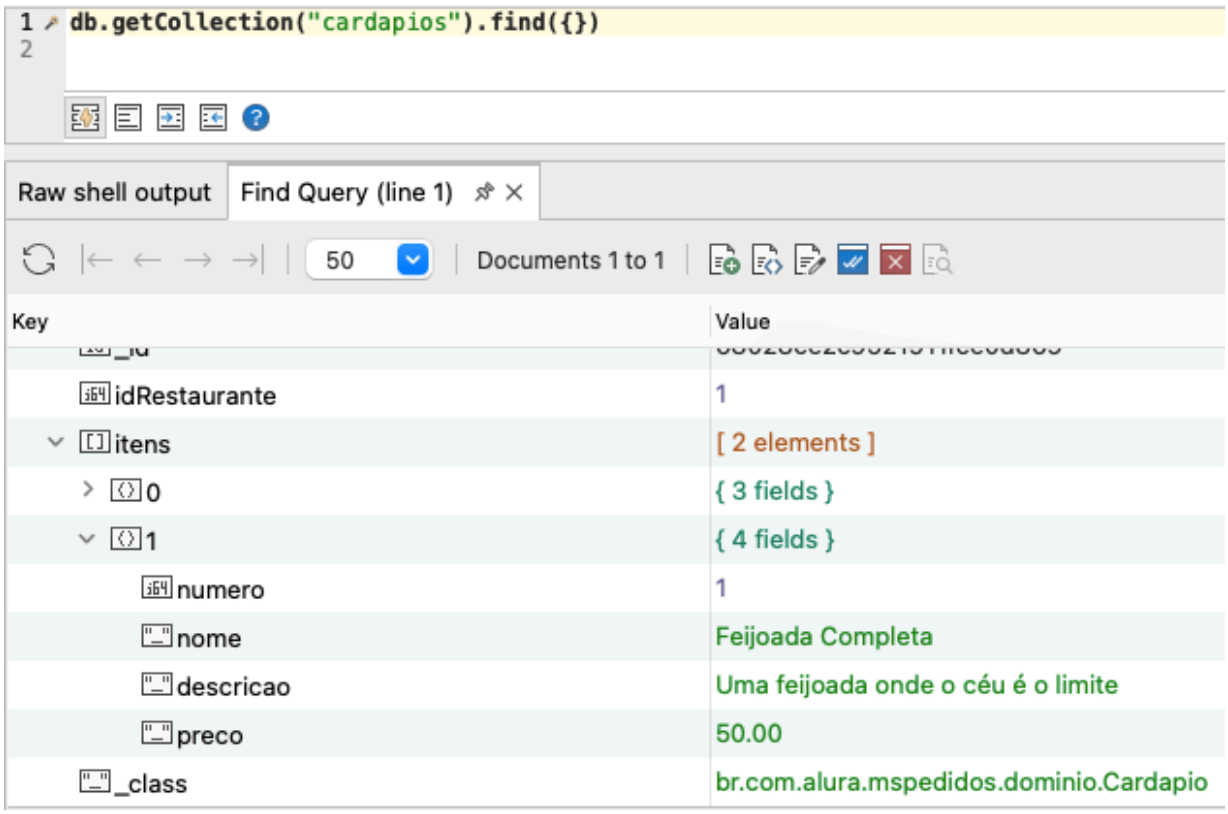


Figura 7.4: Novo item de cardápio no banco de dados.

Inclusão do preço no envio de pedidos realizado pelo *ms-pedidos*

Agora que os itens de cardápio possuem preço, podemos preparar o *ms-pedidos* para enviar pedidos com essa informação.

Na aplicação há apenas inclusões do atributo `preco` em algumas classes, semelhante ao que já fizemos em outras classes da mesma aplicação:

- em `ItemDoPedido` vamos adicionar apenas um método para calcular o preço. Esse método considera o o preço do `ItemCardapio` (já inserido anteriormente) e a quantidade selecionada, resultando na seguinte implementação:

```
public BigDecimal preco(){
    return item.preco().multiply(new
BigDecimal(quantidade));
}
```

Esse método será usado para calcular o total do pedido. Logo, precisamos de um campo com esse nome na classe `Pedido`, fazendo com que ela fique como a seguir:

```
@Document(collection = "pedidos")
@EqualsAndHashCode(exclude = "id")
public class Pedido {
```

```
    @Getter
    @Id
    private String id;
    private Set<ItemDoPedido> itensDoPedido;
    private Endereco endereco;
    private BigDecimal total;

    public Pedido(Set<ItemDoPedido> itensDoPedido,
Endereco endereco) {
        this.itensDoPedido = itensDoPedido;
        this.endereco = endereco;
        this.total = itensDoPedido.stream()
                .map(ItemDoPedido::preco)
                .reduce(BigDecimal.ZERO,
BigDecimal::add);
    }

    public
br.com.alura.mspedidos.infra.kafka.avro.Pedido toAvro()
{
        return
br.com.alura.mspedidos.infra.kafka.avro.Pedido.newBuild
er()
                .setEndereco(endereco.toAvro())

.setItensDoPedido(itensDoPedido.stream().map(ItemDoPedi
do::toAvro).collect(toList()))
                .setTotal(total)
                .build();
    }
}
```

Devemos também incluir o novo campo em `ItemCardapio.avsc`, e usaremos o tipo lógico `decimal`, com precisão de nove casas antes da vírgula e duas

após, fazendo com que o arquivo fique desta maneira:

```
{
  "namespace":
"br.com.alura.mspedidos.infra.kafka.avro",
  "type": "record",
  "doc": "Descreve um item do cardápio. Usado para
descrever detalhadamente um item de pedido",
  "name": "ItemCardapio",
    "fields": [
        {
        "name": "numero",
        "type": "long"
        },
        {
        "name": "nome",
        "type": "string"
        },
        {
        "name": "descricao",
        "type": "string"
        },
        {
        "name": "preco",
          "type": {
            "type": "bytes",
            "logicalType": "decimal",
            "precision": 9,
            "scale": 2
          }
        }
    ]
}
```

Com as alterações realizadas no *ms-pedidos*, precisamos informar ao Schema Registry que temos uma nova versão de *schema* para o tópico `pedidos`. Para isso:

- movemos a lógica de geração de *schema* usando o Avro Schema Merger para um novo *script*, do `pedidos-value-cadastrar.sh` para o `pedidos-value-cadastrar-v2.sh`
- usamos a mesma técnica de atualização de *schema* já usada para o `restaurante.cardapio.itens-value`, no que diz respeito ao modo de compatibilidade. Assim, será possível atualizarmos o *schema* sem problemas. Faremos isso no novo *script bash* nomeado como `pedidos-value-cadastrar-v2.sh`

Com essas alterações, o `pedidos-value-cadastrar.sh` passa a ser mais simples. A seguir, o seu conteúdo:

```
cd ../schemas
curl -X POST http://localhost:8081/subjects/pedidos-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@pedidos-value.js
```

E o `pedidos-value-cadastrar-v2.sh` está na sequência:

```
echo "\nAlterando a compatibilidade do schema pedidos-
value para NONE temporariamente"
```

```
curl -X PUT http://localhost:8081/config/pedidos-value
\
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "NONE"}'

cd ../avroschemamerger
./instalar_avroschemamerger.sh
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-pedidos/src/main/avro true Endereco.avsc
ItemCardapio.avsc ItemDoPedido.avsc Pedido.avsc >
../schemas/pedidos-value-v2.js

cd ../schemas
echo "\nCadastrando a nova versão do schema"
curl -X POST http://localhost:8081/subjects/pedidos-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@pedidos-value-v2.js

echo "\nRetornando a compatilidade do schema
restaurantes.cardapio.itens-value para BACKWARD"
curl -X PUT http://localhost:8081/config/pedidos-value
\
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "BACKWARD"}'
```

Com o *script bash* pronto, podemos referenciá-lo em `infra-up.sh` assim como todos os outros:

```
# trechos de código omitidos por brevidade

while true; do
```

```
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
    ./entregadores-value-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar-v2.sh
    ./pedidos-value-cadastrar-v2.sh
    break
  fi
  sleep 5
done
```

Ao rodarmos o `pedidos-value-cadastrar-v2.sh` e o *ms-pedidos* já com as alterações realizadas, estaremos prontos para testar a utilização do novo campo. A seguir, ambas as execuções:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./pedidos-value-cadastrar-v2.sh

Alterando a compatibilidade do schema pedidos-value
para NONE temporariamente
{"compatibility":"NONE"}
Cadastrando a nova versão do schema
{"id":6}
Retornando a compatilidade do schema
restaurantes.cardapio.itens-value para BACKWARD
{"compatibility":"BACKWARD"}%

cd ..
mvn -f ms-pedidos/pom.xml spring-boot:start
```

Ao enviarmos um novo pedido usando o Postman ou cURL:

```
curl -X POST 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o
limite",
        "preco": 50.00
      },
      "quantidade": 2
    }
  ],
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
'
```

Podemos observar o pedido presente no tópico `pedidos`, já com o preço de cada item e o total do pedido:

Figura 7.5: Novo pedido presente no tópico de pedidos, com o novo campo de preço.

Inclusão do preço no envio de pedidos para os entregadores no *ms-entregadores*

Os pedidos já chegam para os entregadores com o preço de cada item e o seu valor total. Precisamos, portanto, atualizar o *ms-entregadores* para lidar com essa nova informação.

Vamos inicialmente alterar o `Pedido.avdl` para incluir o `preco`. A declaração de tipos lógicos em arquivos `avdl` é análoga à declaração em `avsc`. A seguir, o `Pedido.avdl` já com as alterações:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol PedidoProtocol {

    import idl "Endereco.avdl";
    import idl "ItemDoPedido.avdl";

    record Pedido {
        array<ItemDoPedido> itensDoPedido;
        Endereco endereco;

        @logicalType("decimal")
        @precision(9)
        @scale(2)
        bytes total;
    }
}
```

O `ItemDoPedido.avdl` também deve possuir o seu preço. Podemos nomear o campo da maneira que for mais conveniente. Então, vamos chamá-lo de `precoUnitario`, fazendo com que esse arquivo fique assim:

```
@namespace("br.com.alura.msentregadores.infra.kafka.avro")
protocol ItemDoPedidoProtocol {

    record ItemDoPedido {
        string nome;
        string descricao;
        long numero;
        int quantidade;
```

```
        @logicalType("decimal")
        @precision(9)
        @scale(2)
        bytes precoUnitario;
    }
}
```

Podemos compilar as classes do microsserviço já com os novos campos. Assim, os `stubs` gerados passam a contar com os novos campos e respectivos `getters`, e poderemos usá-los nas classes da aplicação:

```
mvn -f ms-entregadores/pom.xml compile
```

A classe `Pedido` pode ser atualizada para lidar com os novos campos. Tanto o método `fromMsPedidos()` quanto o `toAvro()` podem, respectivamente, ler a nova informação e escrever no objeto a ser enviado para o entregador. Também precisamos inserir o total do pedido como atributo da classe.

Na sequência, a classe com as devidas alterações:

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
@EqualsAndHashCode(exclude = "total")
public class Pedido {

    private Set<ItemDoPedido> itensDoPedido;
    private Endereco endereco;
    private BigDecimal total;

    public static final Pedido NENHUM = new Pedido();
```

```
    public static Pedido
fromMsPedidos(br.com.alura.mspedidos.infra.kafka.avro.P
edido pedidoAvro) {
        Pedido pedido = new Pedido();
        pedido.endereco =
Endereco.fromMsPedidos(pedidoAvro.getEndereco());
        pedido.itensDoPedido =
ItemDoPedido.fromMsPedidos(pedidoAvro.getItensDoPedido(
));
        pedido.total = pedidoAvro.getTotal();
        return pedido;
    }

    public
br.com.alura.msentregadores.infra.kafka.avro.Pedido
toAvro() {
        return
br.com.alura.msentregadores.infra.kafka.avro.Pedido.new
Builder()
                .setEndereco(endereco.toAvro())

.setItensDoPedido(itensDoPedido.stream().map(ItemDoPedi
do::toAvro).collect(toList()))
                .setTotal(total)
                .build();
    }
}
```

A classe `ItemDoPedido` também possui dois métodos com o mesmo nome dos que já foram apresentados. Os ajustes necessários são semelhantes aos feitos em `Pedido` : tratamento do preço em ambos os

métodos e inclusão de atributo `precoUnitario`. A seguir, a classe com as alterações:

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
@EqualsAndHashCode(exclude = "precoUnitario")
public class ItemDoPedido {

    private String nome;
    private String descricao;
    private long numero;
    private int quantidade;
    private BigDecimal precoUnitario;

    public static Set<ItemDoPedido> fromMsPedidos(

List<br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedi
do> itensDoPedidoAvro) {
        return itensDoPedidoAvro.stream()
                .map(ItemDoPedido::toItemDoPedido)
                .collect(Collectors.toSet());
    }

    private static ItemDoPedido toItemDoPedido(

br.com.alura.mspedidos.infra.kafka.avro.ItemDoPedido
itemDoPedidoAvro){
        ItemDoPedido itemDoPedido = new ItemDoPedido();
        itemDoPedido.descricao =
valueOf(itemDoPedidoAvro.getItemCardapio().getDescricao
());
        itemDoPedido.nome =
valueOf(itemDoPedidoAvro.getItemCardapio().getNome());
        itemDoPedido.numero =
itemDoPedidoAvro.getItemCardapio().getNumero();
        itemDoPedido.quantidade =
```

```
itemDoPedidoAvro.getQuantidade();
        itemDoPedido.precoUnitario =
itemDoPedidoAvro.getItemCardapio().getPreco();
        return itemDoPedido;
    }

    public
br.com.alura.msentregadores.infra.kafka.avro.ItemDoPedi
do toAvro(){
       return
br.com.alura.msentregadores.infra.kafka.avro.ItemDoPedi
do.newBuilder()
                .setNome(nome)
                .setDescricao(descricao)
                .setNumero(numero)
                .setQuantidade(quantidade)
                .setPrecoUnitario(precoUnitario)
                .build();
    }
}
```

Após as modificações no código do microsserviço, podemos partir para o *script* de atualização do *schema*, a ser aplicado no Schema Registry.

Assim como fizemos na atualização de `pedidos-value`:

- criamos o `entregadores-value-cadastrar-v2.sh`;
- movemos a lógica de criação automática de *schemas* de `entregadores-value-cadastrar.sh` para esse novo *script*;

- alteramos o modo de compatibilidade conforme a necessidade, para que possamos cadastrar a nova versão do *schema* sem problemas.

O `entregadores-value-cadastrar-v2.sh` deve, portanto, ficar assim:

```
echo "\nAlterando a compatibilidade do schema
entregadores-value para NONE temporariamente"
curl -X PUT http://localhost:8081/config/entregadores-
value \
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "NONE"}'

echo "\nCadastrando a nova versão do schema"

AVRO_TOOLS="avro-tools-1.12.0.jar"
URL="https://repo1.maven.org/maven2/org/apache/avro/avr
o-tools/1.12.0/avro-tools-1.12.0.jar"

if [ ! -f "$AVRO_TOOLS" ]; then
  echo "Avro tools não encontrado, iniciando o
download..."
  wget -O "$AVRO_TOOLS" "$URL"
else
  echo "Avro tools já existe. Realizando a conversão de
avdl em avsc.."
fi

java -jar "$AVRO_TOOLS" idl2schemata ../ms-
entregadores/src/main/avro/Entregador.avdl out

cd ../avroschemamerger
./instalar_avroschemamerger.sh
```

```
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../schemas/out true Entregador.avsc >
../schemas/entregadores-value-v2.js

cd ../schemas

curl -X POST
http://localhost:8081/subjects/entregadores-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@entregadores-value-v2.js

 rm -rf ../schemas/out

echo "\nRetornando a compatilidade do schema
entregadores-value para BACKWARD"
curl -X PUT http://localhost:8081/config/entregadores-
value \
  -H "Content-Type:
application/vnd.schemaregistry.v1+json" \
  --data '{"compatibility": "BACKWARD"}'
```

Assim como todos os outros *scripts* de criação ou atualização de *schemas*, precisamos inserir esse novo *script* em `infra-up.sh` :

```
# trechos de código omitidos por brevidade

while true; do
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
```

```
    ./entregadores-value-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar-v2.sh
    ./pedidos-value-cadastrar-v2.sh
    ./entregadores-value-cadastrar-v2.sh
    break
  fi
  sleep 5
done
```

Ao rodarmos o *script* recém-criado, a atualização é realizada sem que haja problemas:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo schemas %
./entregadores-value-cadastrar-v2.sh

Alterando a compatibilidade do schema entregadores-
value para NONE temporariamente
{"compatibility":"NONE"}
Cadastrando a nova versão do schema
Avro tools já existe. Realizando a conversão de avdl em
avsc..
{"id":7}
Retornando a compatibilidade do schema entregadores-
value para BACKWARD
{"compatibility":"BACKWARD"}%
```

Quando subimos a aplicação, é possível notar que o tópico `entregadores` já possui um entregador. Isso ocorre porque, ao alterarmos o *ms-pedidos*, incluímos um pedido no tópico de `pedidos`. O *ms-entregadores* apenas leu esse pedido, selecionou o entregador e enviou a mensagem para o tópico

`entregadores` com o entregador e seu respectivo pedido.

7.6 Um desserializador de mensagens customizado - obtendo o Schema ID nas mensagens e a última versão de schema de um subject

A arquitetura da nossa solução faz com que exista apenas um produtor de mensagens por tópico presente no Kafka. Logo, quando precisamos inserir o novo atributo `preco`, precisamos apenas nos preocupar com ele e com seus possíveis consumidores.

Mas isso não é verdade em todos os casos. Há arquiteturas nas quais existem diversos produtores de mensagens, cada um postando mensagens criadas com versões diferentes de *schemas*. Isso pode causar problemas nos consumidores que estejam esperando mensagens preparadas com *schemas* mais recentes.

Os consumidores podem, portanto, possuir um mecanismo de auditoria que verifique qual a versão do *schema* usada na criação da mensagem. Dessa forma, os desenvolvedores das aplicações

consumidoras de mensagens terão mais informações para verificar possíveis *bugs* durante o consumo.

Tomemos como exemplo o *ms-entregadores*. Poderíamos implementar essa auditoria na classe `Receptor`, que possui boa parte da lógica de processamento da mensagem. Entretanto, quando a mensagem chega a essa classe, ela já foi convertida para a classe `Pedido`, não sendo mais possível auditá-la, visto que a conversão para o objeto esperado já foi feita, e não é mais possível obter informações relacionadas ao *schema* associado.

Mas, além de `Receptor`, há outra classe que lida com a mensagem: a `KafkaAvroDeserializer`. Essa classe recebe a mensagem como um *array* de *bytes*, converte-a para `Pedido`, e envia o objeto para o `Receptor`. Logo, um bom lugar para inserirmos esse mecanismo seria em `KafkaAvroDeserializer`. No entanto, essa classe não permite alterações dessa natureza.

A solução é criarmos um desserializador customizado que:

- receba a mensagem no formato que ela está armazenada no tópico;
- obtenha as informações do *schema* usado na mensagem;

- converta a mensagem para `Pedido` (que é o objeto esperado pelo `Receptor`);
- envia a mensagem já convertida para o `Receptor`.

> Embora o Redpanda Console nos exiba a mensagem no formato JSON, ela é armazenada como um *array* de *bytes* no tópico. O Redpanda apenas a exibe em um formato amigável para humanos, baseado no *schema* associado à mensagem (previamente cadastrado no Schema Registry). É por isso que precisamos configurar o Schema Registry no Redpanda.

Vamos chamar essa classe de `KafkaAvroAuditDeserializer` e escrevê-la no pacote `br.com.alura.msentregadores.infra.kafka`. Ela precisa implementar a interface `org.apache.kafka.common.serialization.Deserializer<T>`, onde `T` deve ser substituído pela classe cujo objeto abrigará a mensagem desserializada (no caso, o `Pedido`):

```
@Slf4j
public class KafkaAvroAuditDeserializer implements
Deserializer<Pedido> {
}
```

Utilizamos a anotação `@Slf4j` do Lombok para simplificar a implementação de um mecanismo de *log*. No entanto, seu uso não é obrigatório e pode ser substituído por qualquer outra ferramenta de sua preferência.

A *interface* `Deserializer` impõe a implementação de alguns métodos:

- o `public void configure(Map<String, ?> configs, boolean isKey)`, que possui como parâmetros um mapa de configurações e um `boolean` que informa se o desserializador será usado na chave ou no valor da mensagem. Esse método prepara o desserializador com base nas configurações que preparamos no método `consumerConfigs()` da classe `KafkaObjects`;
- o `public Pedido deserialize(String topic, byte[] mensagem)`, que recebe o nome do tópico e a mensagem. Faz a desserialização efetiva dela;
- o `public void close()` que encerra o desserializador. Todos os recursos abertos nos outros métodos devem ser fechados nele.

Leitura do *Schema ID* das mensagens

Para obter as informações do *schema* usado nas mensagens, poderíamos enviar o *schema* nelas mesmas, além da própria mensagem. Assim, os consumidores podem ler o *schema* contido nela e traduzi-la. Entretanto, trabalhar dessa maneira faz com que as mensagens fiquem maiores, o que aumenta a latência e torna as retransmissões mais custosas.

Enviar o *schema* das mensagens diretamente na mensagem pode não ser vantajoso. Mas podemos enviar o *Schema ID* da mensagem em cada uma delas. Como ele identifica unicamente um *schema*, podemos rastrear qual *schema* foi usado para a mensagem.

Felizmente, as implementações de Kafka e Schema Registry inserem essa informação automaticamente na mensagem assim que ela é validada pelo Schema Registry. A Confluent — fabricante das implementações que estamos utilizando — modifica a mensagem acrescentando cinco *bytes* no início, que são utilizados para armazenar as seguintes informações:

- o primeiro *byte*, conhecido como *magic byte*, contém a versão do formato de serialização usado pela Confluent. Atualmente, esse valor é sempre zero.

- Os *bytes* de 1 a 4 (inclusive) são reservados para o *Schema ID*.

> Mais informações a respeito do *Schema ID* podem ser obtidas em https://www.confluent.io/blog/how-schema-registry-clients-work/

A figura ilustra a mensagem após as alterações:

Figura 7.6: Mensagem com alterações.

Como o método `deserialize()` recebe a mensagem como parâmetro no formato *array* de *bytes*, precisamos apenas ler os primeiros *bytes* dele. Usaremos a classe `ByteBuffer`, que realiza a conversão automática dos *bytes* para tipos primitivos:

- enviamos o *array* para um objeto `ByteBuffer` usando o método `wrap()`;
- descartamos o primeiro *byte*, usando o método `get()`;
- obtemos o *Schema ID* como um inteiro, usando o método `getInt()`.

O código para a obtenção do *Schema ID* encontra-se a seguir:

```
    @Override
    public Pedido deserialize(String topic, byte[]
mensagem) {
        final var wrap = ByteBuffer.wrap(mensagem);
        wrap.get(); // descartando o byte 0
        final var schemaId = wrap.getInt(); // obtendo
os 4 bytes referentes ao Schema ID
        log.info("ID do schema extraido da mensagem:
{}", schemaId);
    }
```

Precisamos completar esse método, convertendo o *array* de *bytes* para um objeto da classe `Pedido` e, para isso, podemos usar a classe `KafkaAvroDeserializer`, já utilizada anteriormente. A diferença, neste caso, é que vamos instanciar um objeto dessa classe dentro do nosso desserializador. E, assim como é feito pelas bibliotecas que lidam com o Kafka, precisamos enviar as configurações do método `consumerConfigs()` de `KafkaObjects` para ele.

Para isso, vamos criar um atributo no nosso desserializador:

```
@Slf4j
public class KafkaAvroAuditDeserializer implements
Deserializer<Pedido> {
    private KafkaAvroDeserializer
kafkaAvroDeserializer;
```

```
    // demais códigos
}
```

Em seguida, instanciamos esse atributo no método `configure()` do `KafkaAvroAuditDeserializer`, enviando também os parâmetros do método para o atributo:

```
@Override
public void configure(Map<String, ?> configs, boolean
isKey) {
    kafkaAvroDeserializer = new
KafkaAvroDeserializer();
    kafkaAvroDeserializer.configure(configs, isKey);
}
```

Agora podemos chamar o método `deserialize()` de `KafkaAvroDeserializer` no método `deserialize()` do nosso desserializador:

```
@Override
public Pedido deserialize(String topic, byte[]
mensagem) {
    final var wrap = ByteBuffer.wrap(mensagem);
    wrap.get(); // descartando o byte 0
    final var schemaId = wrap.getInt(); // obtendo os 4
bytes referentes ao Schema ID
    log.info("ID do schema extraido da mensagem: {}",
schemaId);
    return (Pedido)
kafkaAvroDeserializer.deserialize(topic, mensagem);
}
```

Por último, fechamos o `kafkaAvroDeserializer` chamando o seu método `close()`. Esse método será chamado dentro do método de mesmo nome, presente no nosso desserializador:

```
@Override
public void close(){
    kafkaAvroDeserializer.close();
}
```

Agora que implementamos nosso próprio desserializador, precisamos informar o *ms-entregadores* para utilizá-lo em substituição ao `KafkaAvroDeserializer`. Para isso, fazemos a substituição do desserializador no método `consumerConfigs()` de `KafkaObjects`. A seguir, o método após essa alteração:

```
Map<String, Object> consumerConfigs() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroAuditDeserializer.class
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
```

```
IG, schemaRegistryUrl
            ,
KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG
, true
        );
    }
```

Para verificar se o `KafkaAvroAuditDeserializer` está funcionando corretamente, precisamos inicialmente verificar o *Schema ID* da última versão do *subject* `entregadores-value`. Podemos obter essa informação usando o Redpanda Console, ou mesmo usando o cURL:

```
rodrigovieirapinto@MBP-de-Rodrigo codigo-fonte % curl -
X GET http://localhost:8081/subjects/pedidos-
value/versions/latest
{"subject":"pedidos-
value","version":2,"id":6,"schema":"
{\"type\":\"record\",\"name\":\"Pedido\",\"namespace\":
\"br.com.alura.mspedidos.infra.kafka.avro\",\"doc\":\"R
epresenta uma compra de produtos de um mesmo
restaurante\",\"fields\":
[{\"name\":\"itensDoPedido\",\"type\":
{\"type\":\"array\",\"items\":
{\"type\":\"record\",\"name\":\"ItemDoPedido\",\"doc\":
\"Representa um dos muitos produtos que um pedido pode
ter. Descreve também a quantidade de produtos de mesmo
tipo\",\"fields\":[{\"name\":\"itemCardapio\",\"type\":
{\"type\":\"record\",\"name\":\"ItemCardapio\",\"doc\":
\"Descreve um item do cardápio. Usado para descrever
detalhadamente um item de pedido\",\"fields\":
[{\"name\":\"numero\",\"type\":\"long\"},
{\"name\":\"nome\",\"type\":\"string\"},
```

```
{\"name\":\"descricao\",\"type\":\"string\"},
{\"name\":\"preco\",\"type\":
{\"type\":\"bytes\",\"logicalType\":\"decimal\",\"preci
sion\":9,\"scale\":2}}]}},
{\"name\":\"quantidade\",\"type\":\"int\"}]}}},
{\"name\":\"endereco\",\"type\":
{\"type\":\"record\",\"name\":\"Endereco\",\"doc\":\"O
endereço onde o pedido deve ser entregue\",\"fields\":
[{\"name\":\"rua\",\"type\":\"string\"},
{\"name\":\"numero\",\"type\":\"int\"},
{\"name\":\"bairro\",\"type\":\"string\"},
{\"name\":\"cidade\",\"type\":\"string\"},
{\"name\":\"estado\",\"type\":\"string\"},
{\"name\":\"cep\",\"type\":\"string\"},
{\"name\":\"complemento\",\"type\":
[\"null\",\"string\"],\"default\":null}]}},
{\"name\":\"total\",\"type\":
{\"type\":\"bytes\",\"logicalType\":\"decimal\",\"preci
sion\":9,\"scale\":2}}]}"}%
```


A informação que queremos está no campo `id` da mensagem 6, neste caso.

Ao rodarmos o *ms-entregadores* e enviarmos um pedido via *ms-pedidos*, é possível notar que o *Schema ID* impresso no *log* é o mesmo que obtemos do Schema Registry:

```
2025-04-19T16:19:28.329-03:00  INFO 11657 --- [ms-
entregadores] [ntainer#0-0-C-1]
b.c.a.m.i.k.KafkaAvroAuditDeserializer   : ID do schema
extraido da mensagem: 6
```

Obtendo a última versão do *schema* de um *subject*

A interface `SchemaRegistryClient` pertence a uma das dependências da biblioteca `kafka-avro-serializer` e possui diversos métodos que permitem diversas interações com o Schema Registry. Por exemplo:

- o `getAllSubjects()` retorna todos os *subjects* cadastrados no Schema Registry;
- o `getAllVersions(String subject)` retorna todas as versões de *schemas* de um *subject*;
- o `getCompatibility(String subject)` informa o modo de compatibilidade do *subject* passado como parâmetro.

Há também o método `getLatestSchemaMetadata(String subject)` que nos traz a última versão de *schema* de um *subject*. E podemos usar esse método no `KafkaAvroAuditDeserializer` para obter essa informação do Schema Registry.

> Note que *Schema ID* e versão de *schema* são informações diferentes. Enquanto o *Schema ID* identifica unicamente um *schema* dentro do Schema Registry, a versão identifica os diferentes *schemas* cadastrados dentro de um *subject*.

Para usarmos o `SchemaRegistryClient`, vamos introduzi-lo inicialmente como atributo do nosso desserializador:

```
@Slf4j
public class KafkaAvroAuditDeserializer implements
Deserializer<Pedido> {

    private KafkaAvroDeserializer
kafkaAvroDeserializer;
    private SchemaRegistryClient schemaRegistryClient;

// demais códigos
}
```

Em seguida, vamos instanciar o novo atributo. E uma implementação de `SchemaRegistryClient` é o `CachedSchemaRegistryClient`, que possui um dos construtores com a seguinte assinatura:

```
public CachedSchemaRegistryClient(String baseUrl, int
cacheCapacity)
```

O primeiro parâmetro representa a URL do Schema Registry, enquanto o segundo é a quantidade de *schemas* que podem ser armazenados em cache.

Podemos instanciar esse atributo assim como instanciamos um objeto de `KafkaAvroDeserializer`: diretamente no `KafkaAvroAuditDeserializer`. Mas como a URL do Schema Registry é obtida na classe

`KafkaObjects`, vamos realizar a instância usando o método a seguir, em `KafkaObjects`:

```
private SchemaRegistryClient schemaRegistryClient(){
    return new
CachedSchemaRegistryClient(schemaRegistryUrl, 100);
}
```

Passamos a instância como uma das configurações presentes em `consumerConfigs()` de `KafkaObjects`. Usando também o mapa de configurações, podemos adicionar o nome do *subject*. Esse nome é, basicamente, o nome do tópico acrescido do sufixo "-value", e é a informação que o método `getLatestSchemaMetadata()` de `SchemaRegistryClient` espera.

O mapa de configurações com essas alterações encontra-se a seguir:

```
Map<String, Object> consumerConfigs() {
    return Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl
            ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class
            ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroAuditDeserializer.class
            , ConsumerConfig.GROUP_ID_CONFIG, grupo
            , ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
```

```
            ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
            ,
KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG
, true
            , SCHEMA_REGISTRY_CLIENT,
schemaRegistryClient()
            , SCHEMA_NAME, topicoDePedidos + "-value"
        );
    }
```

As chaves `SCHEMA_REGISTRY_CLIENT` e `SCHEMA_NAME` foram criadas em `KafkaObjects`, e são bem simples:

```
public static final String SCHEMA_REGISTRY_CLIENT =
"schema.registry.client";
public static final String SCHEMA_NAME = "schema.name";
```

Podemos acessar essas configurações recém-inseridas no método `configure()` de `KafkaAvroAuditDeserializer`:

```
@Override
public void configure(Map<String, ?> configs, boolean
isKey) {
    kafkaAvroDeserializer = new
KafkaAvroDeserializer();
    kafkaAvroDeserializer.configure(configs, isKey);
    schemaRegistryClient = (SchemaRegistryClient)
configs.get(KafkaObjects.SCHEMA_REGISTRY_CLIENT);
    schemaName =
(String)configs.get(KafkaObjects.SCHEMA_NAME);
}
```

Agora podemos obter a última versão do *schema* `pedidos-value`. O código para isso envolve algumas alterações no método `deserialize()` de `KafkaAvroAuditDeserializer`, mostradas a seguir:

```
@Override
public Pedido deserialize(String topic, byte[]
mensagem) {
    final var wrap = ByteBuffer.wrap(mensagem);
    wrap.get(); // descartando o byte 0
    final var schemaId = wrap.getInt(); // obtendo os 4
bytes referentes ao Schema ID
    log.info("ID do schema extraido da mensagem: {}",
schemaId);

    try {
        SchemaMetadata schemaMetadata =
schemaRegistryClient.getLatestSchemaMetadata(schemaName
);
        final var version =
schemaMetadata.getVersion();
        log.info("Versão do último schema obtido do
schema registry: {}", version);
        return (Pedido)
kafkaAvroDeserializer.deserialize(topic, mensagem);
    } catch (RestClientException | IOException e) {
        throw new RuntimeException(e);
    }
}
```

O método `getLatestSchemaMetadata()` pode lançar as exceções descritas no bloco `catch`, por isso sua necessidade.

Ao rodarmos novamente o *ms-entregadores* e enviar um novo pedido, podemos ver o log atualizado com as novas informações:

```
2025-04-19T17:23:35.731-03:00  INFO 10986 --- [ms-
entregadores] [ntainer#0-0-C-1]
b.c.a.m.i.k.KafkaAvroAuditDeserializer   : ID do schema
extraido da mensagem: 6
2025-04-19T17:23:35.998-03:00  INFO 10986 --- [ms-
entregadores] [ntainer#0-0-C-1]
b.c.a.m.i.k.KafkaAvroAuditDeserializer   : Versão do
último schema obtido do schema registry: 2
```

O `KafkaAvroAuditDeserializer` implementa, portanto, o padrão de projeto Decorator, visto que ele adiciona responsabilidades adicionais ao desserializador da mensagem.

Para mais informações sobre esse padrão, consulte o livro *Padrões de projetos: Soluções reutilizáveis de software orientado a objetos* (2006).

7.7 A propriedade USE_LATEST_VERSION e boas práticas associadas

Conforme vimos em capítulos anteriores, essa propriedade faz com que tanto o produtor quanto o consumidor utilizem a última versão de *schema* cadastrado no Schema Registry. Seu valor *default* é `false`.

Para que ela seja usada e configurada como `true`, é importante que os desenvolvedores dos produtores e consumidores estejam combinados de que mudanças ocorrerão nos *schemas* utilizados pelas aplicações. Caso contrário, podemos ter cenários em que o *schema* atualizado pelo produtor cause problemas para os consumidores que não possuem as últimas atualizações.

Felizmente, todos os microsserviços dessa solução são produzidos pelo mesmo time da Alura Eats, e todas as alterações de *schemas* ao longo do processo de desenvolvimento foram feitas de forma sincronizada, o que não trouxe maiores problemas. Entretanto, se os produtores e consumidores são mantidos por times ou empresas diferentes, é aconselhável usar essa configuração como `false`.

7.8 Resumo

Neste capítulo, vimos diversas características importantes que envolvem produtores e consumidores de mensagens, bem como alguns novos *endpoints* do Schema Registry. Inserimos novos campos nas mensagens, atualizamos versões de *schema*, trabalhamos com desserializadores customizados, estudamos algumas curiosidades e algumas boas práticas.

No próximo capítulo, trabalharemos com o formato *Protobuf* para envio e recebimento de mensagens. Dessa forma, nossa solução passará a trabalhar com os três principais formatos de dados que as empresas utilizam.

CAPÍTULO 8

O Protobuf como formato de mensagens

Com os pedidos feitos e os entregadores devidamente selecionados, podemos preparar uma solução que informe aos restaurantes de que os entregadores já estão a caminho. E é isso que faremos neste capítulo.

Para esse fim, usaremos o Protobuf como protocolo de mensagens.

> Os códigos presentes aqui encontram-se na *branch* `cap-8` do projeto.

8.1 O formato Protobuf

O *Protocol Buffers* (também conhecido como Protobuf) é um mecanismo de serialização de dados estruturados criado pelo Google, projetado para ser eficiente, compacto e independente de linguagem. Ele permite a definição clara de estruturas de dados (*schemas*) usando uma linguagem específica (.proto), a partir da qual pode ser gerado código-

fonte automaticamente em diversas linguagens, como Java, C++, Python, entre outras.

Sua sintaxe não deve causar muita estranheza para programadores acostumados com Java ou C++, visto que é bem parecida com essas linguagens.

Por exemplo: imagine que precisamos lidar com o conceito de *empregado* dentro da linguagem Java. Um empregado pode ter um nome e um crachá.

Em Java, a classe `Empregado` seria um arquivo chamado `Empregado.java` e, com esses atributos, ficaria dessa forma:

```
package br.com.exemplo;

class Empregado {

    private String nome;
    private int cracha;

    // demais códigos omitidos por brevidade
}
```

Já em Protobuf, podemos escrever um arquivo `empregado.proto` (com letras minúsculas mesmo) e o código ficará assim:

```
syntax = "proto3";

package br.com.exemplo;
```

```
message Empregado {
  string nome = 1;
  int32 cracha = 2;
}
```

A instrução `syntax` informa ao compilador que estamos usando a versão 3 do Protobuf. Já a `package`, assim como no Java, ajuda a organizar o código. A instrução `message` define uma estrutura de dados, semelhante a uma classe Java e seus atributos. Por último, temos os campos `nome` e `cracha` com seus tipos e um número no final de cada campo. Esses são identificadores únicos dos campos no formato binário.

> Ao contrário do que possa parecer, a identificação dos campos na geração das mensagens não é feita pelos seus nomes, mas pelos números declarados no final de cada campo. Dessa forma, o protocolo pode criar mensagens extremamente compactas e eficientes, visto que o número ocupa menos espaço na mensagem do que o nome. O nome existe no arquivo apenas para facilitar sua identificação pelos desenvolvedores.

O `int32` é um tipo inteiro com tamanho equivalente ao tipo `int` do Java. Há também o `int64`, equivalente ao tipo `long`.

Podemos incluir novos campos em `empregado.proto`. Um empregado pode possuir também informações como data de admissão e um salário. Campos como esses são melhor representados com tipos específicos, assim como fazemos no Java (em que usamos respectivamente `LocalDate` e `BigDecimal`).

Com esses novos campos, a classe `Empregado` deve ficar assim:

```
class Empregado {
    private String nome;
    private int cracha;
    private LocalDate dataAdmissao;
    private BigDecimal salario;
}
```

Entretanto, diferente do Avro, que possui os tipos lógicos para representar campos como esses, o Protobuf possui diversos tipos criados pelo Google que permitem uma melhor representação da informação.

- para representar a data de admissão, podemos usar o `google.type.DateTime`, que encapsula uma data e hora;
- para o salário, podemos usar o `google.type.Decimal`.

Para inserirmos esses novos tipos, precisamos importar essas classes em `empregado.proto`. Os

`imports` devem ser declarados como a seguir:

```
import "google/type/datetime.proto";
import "google/type/decimal.proto";
```

E os atributos, assim:

```
  google.type.DateTime dataAdmissao = 3;
  google.type.Decimal salario = 4;
```

O `empregado.proto` fica da seguinte forma, após essas alterações:

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

package br.com.exemplo;

message Empregado {
  string nome = 1;
  int32 cracha = 2;
  google.type.DateTime dataAdmissao = 3;
  google.type.Decimal salario = 4;
}
```

Uma descrição completa do protocolo, bem como o suporte a diversas linguagens de programação, encontra-se em https://protobuf.dev/getting-started/

A documentação de todos os tipos criados pelo Google pode ser consultada em https://cloud.google.com/java/docs/reference/proto-google-common-protos/latest/overview

Veremos a seguir como usar esses novos tipos na nossa solução, bem como outros recursos do Protobuf.

8.2 Notificando os restaurantes do despacho dos entregadores aos destinos: o ms-notificacoes

Conforme vimos no capítulo 6, assim que um entregador é selecionado, é enviada uma mensagem ao tópico `entregadores`, contendo informações sobre o próprio entregador e o pedido que ele vai levar.

O *ms-notificacoes* lê esse tópico e informa que o pedido já se encontra em processo de entrega a todos os sistemas interessados. Como os

restaurantes precisam saber essa informação, o *ms-restaurantes* é uma dessas aplicações.

A integração entre essas aplicações com os tópicos relacionados encontra-se na figura a seguir:

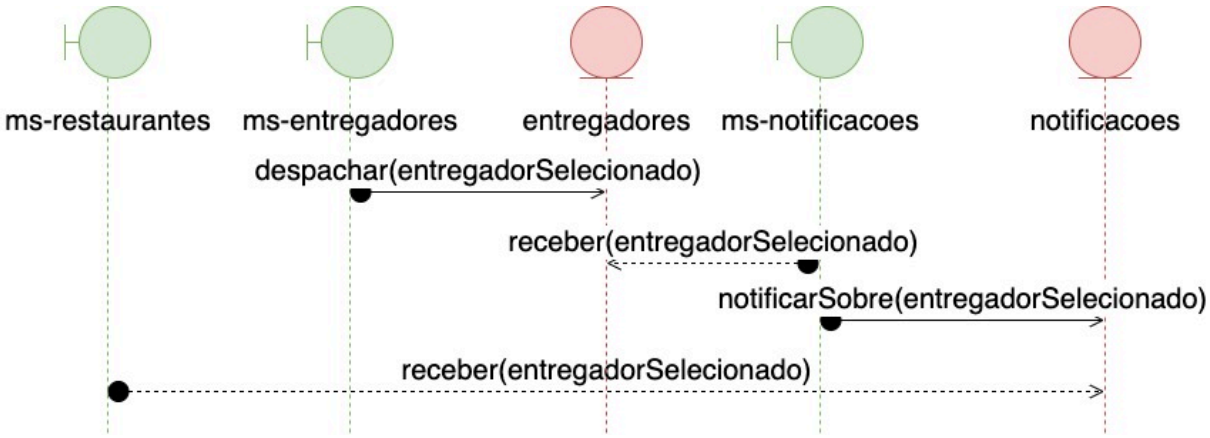


Figura 8.1: Integração entre o ms-pedidos e o ms-entregadores.

Vamos inicialmente criar a infraestrutura necessária, incluindo no `docker-compose.yml` a criação do tópico `notificacoes` juntamente da criação dos demais tópicos:

```
kafka-topics --create --bootstrap-server kafka:29092 --
partitions 1 --replication-factor 1 --topic
notificacoes
```

Precisamos também inserir o suporte à desserialização de mensagens no formato Protobuf no Redpanda Console. Para isso, precisamos informar a ele:

- o Schema Registry que deve ser usado para desserialização (informação já existente).

E ainda devemos:

- habilitar o formato Protobuf na ferramenta, usando a configuração `protobuf.enabled: true`;
- habilitar o Schema Registry para ser usado com o formato Protobuf, com a configuração `protobuf.schemaRegistry.enabled: true`;
- cadastrar um tempo para *refresh* da ferramenta. Assim que esse tempo se esgotar, o Redpanda buscará novos *schemas* no Schema Registry. Usar a configuração `protobuf.schemaRegistry.refreshInterval: 5s` (ou seja, a busca será realizada a cada 5 segundos).

A configuração completa encontra-se adiante:

```
CONSOLE_CONFIG_FILE: |
  kafka:
    brokers: ["kafka:29092"]
    schemaRegistry:
      enabled: true
      urls: ["http://schema-registry:8081"]
    protobuf:
      enabled: true
      schemaRegistry:
        enabled: true
        refreshInterval: 5s
```

O *ms-notificacoes* é uma aplicação simples, que apenas recebe o entregador com seu respectivo pedido, converte a mensagem correspondente para o formato Protobuf e a envia para o tópico `notificacoes`. Para isso, ele usa um receptor como o seguinte:

```
package br.com.alura.msnotificacoes.infra.kafka;

import br.com.alura.msnotificacoes.dominio.Entregador;
import br.com.alura.msnotificacoes.dominio.Notificador;
import lombok.AllArgsConstructor;
import
org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
@AllArgsConstructor
class Receptor {

    private Notificador notificador;

    @KafkaListener(topics = "${topico.entregadores}"
            , groupId = "${spring.kafka.consumer.group-
id}"
            , containerFactory = "kafkaListener")
    public void
receber(br.com.alura.msentregadores.infra.kafka.avro.En
tregador entregadorAvro){


notificador.notificarDespachoDe(Entregador.fromMsEntreg
adores(entregadorAvro));
```

```
    }
}
```

O `Entregador` é uma classe de domínio que, por não possuir muita lógica de negócio, é utilizada para realizar a conversão de si mesma para Protobuf. Seu código é o seguinte:

```
@NoArgsConstructor(access = AccessLevel.PRIVATE)
@EqualsAndHashCode(of = "nome")
public class Entregador {

    private String nome;
    private Pedido pedido;

    public static Entregador
fromMsEntregadores(br.com.alura.msentregadores.infra.ka
fka.avro.Entregador entregadorAvro){
        var entregador = new Entregador();
        entregador.nome =
valueOf(entregadorAvro.getNome());
        entregador.pedido =
Pedido.fromMsEntregadores(entregadorAvro.getPedido());

        return entregador;
    }

    public
br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador
toProtobuf() {
        return
br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador
.newBuilder()
                .setNome(nome)
                .setPedido(pedido.toProtobuf())
```

```
                .build();
    }
}
```

Repare que essa classe possui, em vez de construtores, um método fábrica de construção a partir do objeto Avro, o `fromMsEntregadores()`, semelhante ao que foi feito em capítulos anteriores.

O `Notificador` usado pelo `Receptor` é uma interface, implementada pela classe `NotificadorKafka`:

```
package br.com.alura.msnotificacoes.infra.kafka;

import br.com.alura.msnotificacoes.dominio.Entregador;
import br.com.alura.msnotificacoes.dominio.Notificador;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class NotificadorKafka implements Notificador {

    private KafkaTemplate<Void,
br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador
> entregadorKafkaTemplate;
    private String topico;

    public NotificadorKafka(
            KafkaTemplate<Void,
br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador
> entregadorKafkaTemplate,
            @Value("${topico.notificacoes}") String
```

```
topico){
        this.entregadorKafkaTemplate =
entregadorKafkaTemplate;
        this.topico = topico;
    }

    @Override
    public void notificarDespachoDe(Entregador
entregador) {
        entregadorKafkaTemplate.send(topico,
entregador.toProtobuf());
    }
}
```

A classe `br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador` e todos os seus atributos são criadas por meio da compilação do arquivo `entregador.proto`, presente na solução. Ele possui todas as demais estruturas de dados do entregador, como `Pedido`, `ItemDoPedido` e `Endereco`. Para declararmos um atributo do tipo `Pedido` para o `Entregador`, fazemos algo semelhante ao Java: simplesmente escrevemos o tipo e o nome do atributo, juntamente do identificador único do atributo. A estrutura `Entregador`, portanto, é assim:

```
message Entregador {
  string nome = 1;
  Pedido pedido = 2;
}
```

Há diversos itens de pedido em `Pedido`. E para declararmos essa característica no Protobuf usamos a palavra-chave `repeated`. Para declararmos os itens de pedido, escrevemos:

```
repeated ItemDoPedido itensDoPedido = 1;
```

Os pedidos são enviados com o seu valor total. E, para declarar esse atributo, usaremos o tipo `google.type.Decimal`, já mencionado anteriormente. Assim, não corremos o risco de perder precisão na informação. Para declará-lo, fazemos:

```
google.type.Decimal total = 3;
```

Além disso, precisamos que o pedido seja enviado para o tópico `notificacoes` com uma **data de entrega**. Dessa forma, os restaurantes poderão controlar o momento em que o pedido saiu para o destino. Logo, precisamos de um atributo `dataEntrega` no `Pedido`. E, como já vimos, podemos usar o `DateTime` do pacote `google.type`.

O atributo `dataEntrega` fica assim declarado:

```
google.type.DateTime dataEntrega = 4;
```

O `Pedido`, portanto, deve ser declarado da seguinte forma:

```
message Pedido {
  repeated ItemDoPedido itensDoPedido = 1;
  Endereco endereco = 2;
  google.type.Decimal total = 3;
  google.type.DateTime dataEntrega = 4;
}
```

Há também a palavra-chave `optional`, que informa que um determinado atributo não possui obrigatoriedade no seu preenchimento. Esse recurso é importante para o atributo `complemento` de `Endereco`.

Por último, podemos especificar configurações ou comportamentos adicionais para o compilador, utilizando a palavra-chave `option`. Por exemplo:

```
option java_multiple_files = true;
option java_package =
"br.com.alura.msnotificacoes.infra.kafka.grpc";
```

A opção `java_multiple_files` informa ao compilador que desejamos que seja gerada uma classe Java separada para cada mensagem, em vez de uma só com todas. Já a `java_package` define qual será o pacote das classes que serão geradas.

Ao contrário dos *schemas* Avro ou JSON, que geram classes Java a partir de arquivos nesses formatos com o auxílio de ferramentas, o Protobuf exige a compilação do arquivo `proto`, que possui regras de sintaxe, validações e outras características, tendo como saída o código fonte na linguagem que desejamos (Java, Go etc.).

O arquivo `entregador.proto` completo está a seguir:

```
syntax = "proto3";

package br.com.alura.msnotificacoes.infra.kafka.grpc;

import "google/type/datetime.proto";
import "google/type/decimal.proto";

option java_multiple_files = true;
option java_package =
"br.com.alura.msnotificacoes.infra.kafka.grpc";

message Entregador {
  string nome = 1;
  Pedido pedido = 2;
}

message Pedido {
  repeated ItemDoPedido itensDoPedido = 1;
  Endereco endereco = 2;
  google.type.Decimal total = 3;
```

```
  google.type.DateTime dataEntrega = 4;
}

message ItemDoPedido {
  string nome = 1;
  string descricao = 2;
  int64 numero = 3;
  int32 quantidade = 4;
  google.type.Decimal precoUnitario = 5;
}

message Endereco {
  string rua = 1;
  int32 numero = 2;
  optional string complemento = 3;
  string bairro = 4;
  string cep = 5;
  string cidade = 6;
  string estado = 7;
}
```

8.3 Compilação de arquivos .proto usando o utilitário protoc

Com o `entregador.proto` pronto, podemos compilá-lo, gerando as classes Java como saída. Há algumas maneiras de fazer isso. Uma delas é por meio do `protoc` (*Protocol Buffer Compiler*), um utilitário de linha de comando criado pelo Google.

Para realizar a compilação usando o `protoc`, precisamos inicialmente obter os tipos `DateTime` e `Decimal`, criados pelo Google. Eles podem ser baixados por meio do projeto Google APIs, que contém esses e muitos outros tipos, nas seguintes URLs:

- https://github.com/googleapis/googleapis/blob/master/google/type/datetime.proto
- https://github.com/googleapis/googleapis/blob/master/google/type/decimal.proto

Vamos inseri-los na pasta `ms-notificadores/src/main/proto-exemplo/src/google/type`. Também vamos criar uma outra pasta `saida`, que armazena as classes geradas após a compilação, bem como o `entregador.proto`:

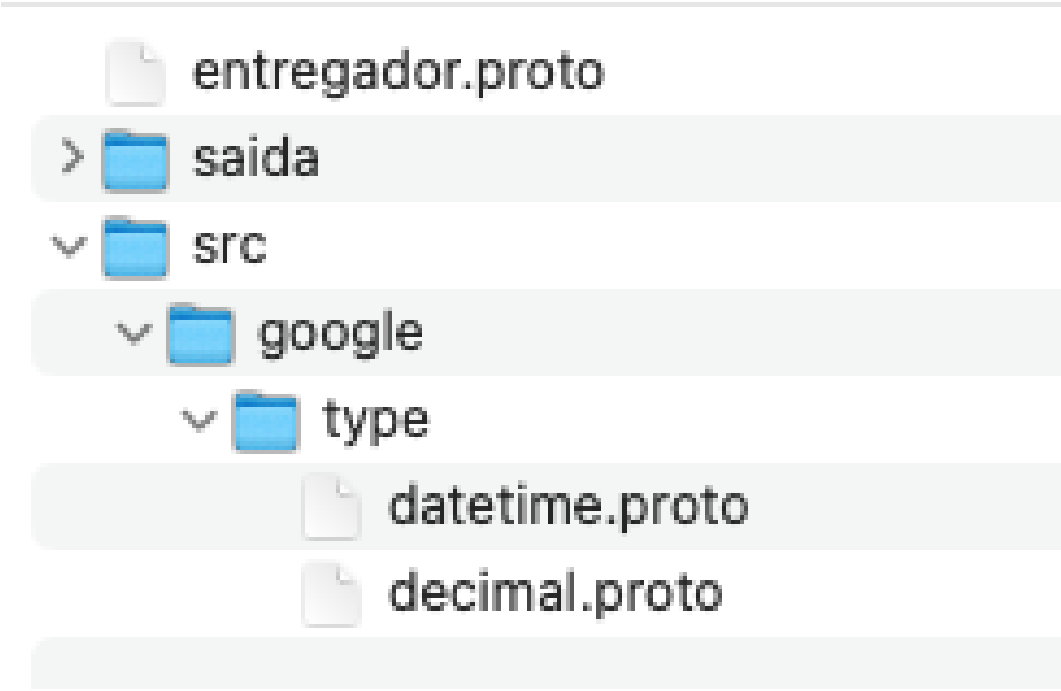


Figura 8.2: Estrutura de pastas para compilação.

O `protoc` pode receber diversos argumentos. A seguir estão alguns deles:

- `--proto_path` : indica a pasta onde se encontram os arquivos `.proto` necessários para a compilação
- `--java_out` : possui dupla função: informa qual deve ser a pasta de saída do resultado da compilação e a linguagem dos códigos de saída (Java, no caso)

Para compilarmos o `entregador.proto` , escrevemos o seguinte comando:

```
protoc --proto_path=. --proto_path=src --java_out=saida
entregador.proto
```

E as classes Java estarão disponíveis na pasta `saida` :

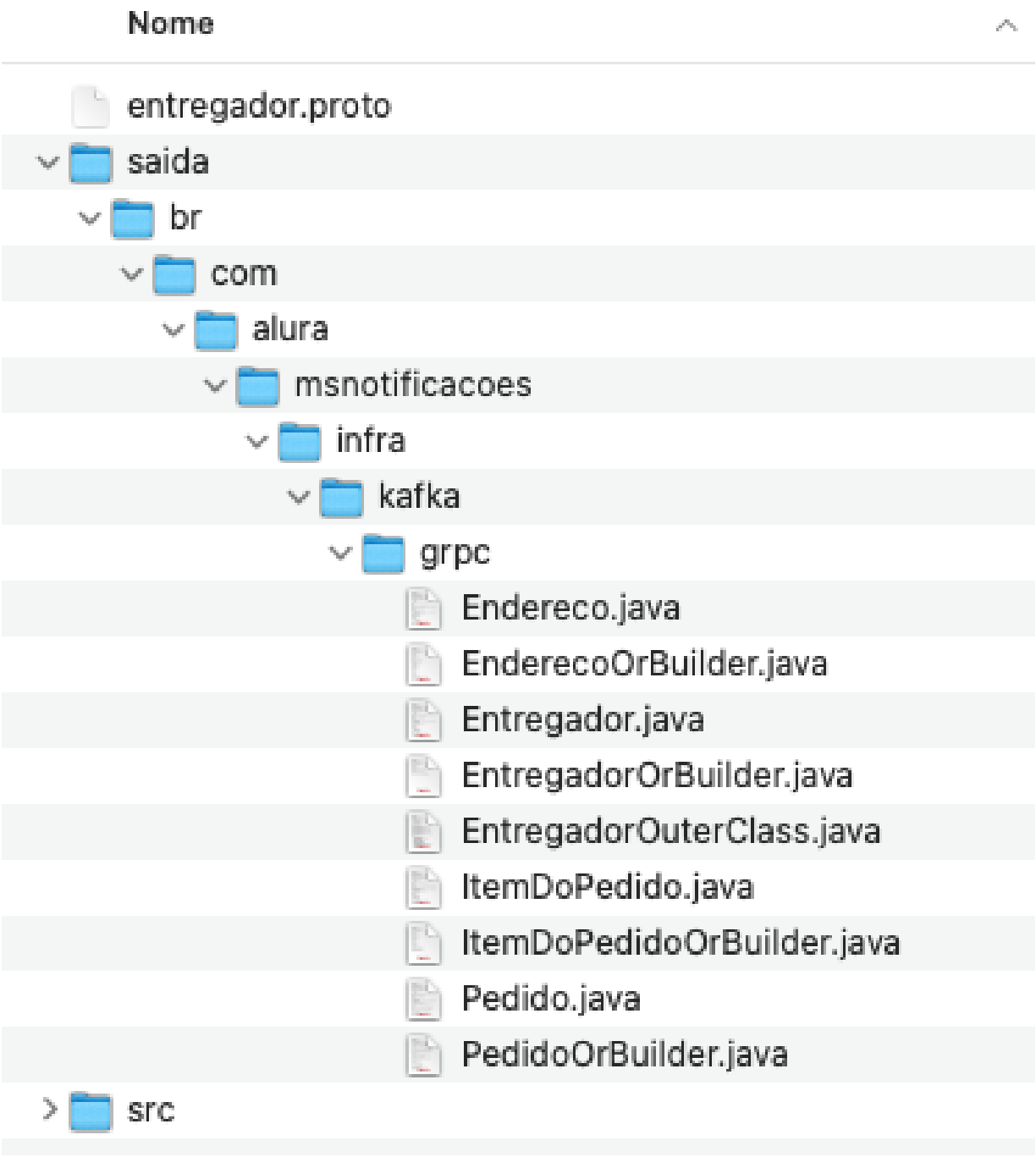


Figura 8.3: Pasta de saída contendo as classes como resultados da compilação.

8.4 O plugin protobuf-maven-plugin e a compilação usando Maven

Embora compilar os arquivos `.proto` usando o `protoc` seja uma tarefa simples, por estarmos trabalhando com um projeto Maven, podemos usar os seus *plugins* para integrarmos o processo de compilação desses arquivos ao ciclo de *build* da ferramenta, facilitando esse processo.

Para isso, usaremos o `protobuf-maven-plugin`, que:

- por padrão, considera todos os arquivos `.proto` presentes na pasta `src/main/protobuf` como elegíveis para compilação;
- recebe como parâmetro apenas a versão de Protobuf usada pela aplicação.

Sua configuração completa encontra-se adiante. Usaremos a versão 3.25.6 do Protobuf:

```
<plugin>
    <groupId>io.github.ascopes</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <configuration>
        <protocVersion>${protobuf-java.version}
</protocVersion>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
```

```
            </goals>
        </execution>
    </executions>
</plugin>
```

Após copiarmos o `entregador.proto` para a pasta `src/main/protobuf` e rodarmos o comando:

```
mvn compile
```

podemos ver o *log* exibindo a compilação do `entregador.proto`:

```
[INFO]
[INFO] --- protobuf:3.1.3:generate (default) @ ms-
notificacoes ---
[INFO] Attempting to resolve artifact
com.google.protobuf:protoc:exe:osx-x86_64:3.25.6
[INFO] Registering
/Users/rodrigovieirapinto/livro/codigo-fonte/ms-
notificacoes/target/generated-sources/protobuf as a
Maven main source root
[INFO] All sources will be compiled, as no previous
build data was detected
[INFO] Generating source code from 1 proto source file
and 0 descriptor files (discovered a total of 1 proto
source file and 0 descriptor files)
[INFO] Invoking protoc (enable debug logs for more
details)
[INFO] protoc (pid 5442) returned exit code 0 (success)
after 163ms
[INFO] PROTOC SUCCEEDED: Protoc invocation succeeded.
```

Para outras informações sobre esse plugin, acesse https://github.com/ascopes/protobuf-maven-plugin

8.5 O KafkaProtobufSerializer

De forma análoga ao `KafkaAvroSerializer` e ao `KafkaJsonSchemaSerializer`, para realizar a serialização para o formato Protobuf, usamos o `KafkaProtobufSerializer`. Ele pertence à biblioteca `kafka-protobuf-serializer`, que deve ser declarada no `pom.xml` como a seguir:

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-protobuf-serializer</artifactId>
    <version>${confluent.version}</version>
</dependency>
```

Assim como nos demais serializadores, usaremos a versão 7.9.1, que corresponde à versão do Kafka que estamos usando.

A classe `KafkaObjects` do *ms-notificacoes* recebe também um *bean* do `KafkaTemplate` a ser usado pelo

`Notificador`, além do mapa de configurações, que deve conter o `KafkaProtobufSerializer`. Eles estão a seguir:

```
@Bean
KafkaTemplate<Void,
br.com.alura.msnotificacoes.infra.kafka.grpc.Entregador
> kafkaTemplate(){
    return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(produtorConfigs()));
}

private Map<String, Object> produtorConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaServerUrl);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
VoidSerializer.class);

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaProtobufSerializer.class);


props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTR
Y_URL_CONFIG, schemaRegistryUrl);

props.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
SCHEMAS, true);

props.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
SION, true);

    return props;
}
```

Como não vamos enviar chaves com as mensagens, usaremos o `VoidSerializer` como serializador de chaves.

8.6 Suporte do Avro Schema Merger a arquivos .proto

Assim como fizemos em todos os tópicos até o momento, é necessário o cadastro do *schema* no Schema Registry para o tópico `notificacoes`.

O JSON que enviamos para o Schema Registry pode possuir o seguinte formato:

```
{
  "schemaType": "JSON",
  "schema": "{\"type\":\"object\",\"properties\":
{\"nome\":{\"type\":\"string\"}},\"required\":
[\"nome\"]}"
}
```

O arquivo é semelhante ao que geramos no capítulo 3, e podemos afirmar que o formato do *schema* a ser cadastrado é JSON (devido ao `schemaType`). O elemento `schema` contém o *schema* a ser cadastrado de fato.

Dissemos que o JSON *pode possuir* o formato mostrado pois o elemento `schemaType` não é obrigatório. Caso não usarmos esse elemento no arquivo, o Schema Registry assume que estamos cadastrando um *schema* Avro. E a construção do Avro Schema Merger foi feita com base nessa premissa. Logo, os *schemas* gerados por ele não escrevem esse elemento.

Mas podemos estender nossa aplicação para suportar também a escrita de arquivos para cadastrar *schemas* no formato Protobuf (ou mesmo JSON). Assim, podemos automatizar a criação dos arquivos de cadastro de *schemas*, sejam eles no formato Avro ou em outros formatos suportados pelo Schema Registry, além de inserir o elemento `schemaType` quando necessário.

Para isso, precisamos inicialmente realizar pequenas alterações na classe `AvroSchemaMerger`. O método `merge()` antes cuidava apenas de arquivos `.avsc`. Agora passará a lidar com arquivos de *schema* em diversos formatos. Logo, vamos apenas alterar algumas variáveis e mensagens informativas, sem alterar sua lógica, como a seguir:

```
public final String merge(File pastaRaiz, boolean
comSchema, List<String> arquivosDaPastaEmOrdem) throws
IOException {
```

```
    List<File> schemas =
arquivosDaPastaEmOrdem.stream()
        .map(nomeArquivoSchema -> new File(pastaRaiz,
nomeArquivoSchema))
        .toList();

    if (schemas == null || schemas.isEmpty()) {
        System.out.println("Nenhum arquivo de schema
(.avsc, .proto) encontrado.");
        return "";
    }
```

Também precisamos de uma solução que retorne o valor a ser inserido no elemento `schemaType`, com base na extensão dos arquivos de *schema* recebidos pelo `AvroSchemaMerger`. Normalmente uma solução dessas passa por diversos `if`, sendo um para cada tipo de *schema*. Ou escrevemos um `enum` como o seguinte:

```
package br.com.alura.avroschemamerger;

import org.apache.commons.io.FilenameUtils;

import java.io.File;
import java.util.Arrays;
import java.util.List;

public enum SchemaType {

    AVRO("avsc"),
    PROTOBUF("proto"),
    JSON("json");
```

```
    private String extensaoArquivo;

    private SchemaType(String extensaoArquivo){
        this.extensaoArquivo = extensaoArquivo;
    }

    public static SchemaType selecionar(List<File>
schemas){
        var extensao =
FilenameUtils.getExtension(schemas.getFirst().getName()
);
        return Arrays
                .stream(values())
                .filter(schemaType ->
schemaType.extensaoArquivo.equalsIgnoreCase(extensao))
                .findFirst().orElse(SchemaType.AVRO);
    }
}
```

O método `selecionar()` recebe os *schemas* como parâmetro e retorna o *schema type* a ser usado.

Podemos agora utilizá-lo em `AvroSchemaMerger`. Há dois lugares dessa classe onde ele é bastante útil:

- na verificação da quantidade de *schemas* recebidos no método `merge()`;
- na verificação do valor da variável `comSchema` recebida como parâmetro do mesmo método.

A classe com as alterações encontra-se na sequência:

```
public class AvroSchemaMerger {

    public final String merge(File pastaRaiz, boolean
comSchema, List<String> arquivosDaPastaEmOrdem) throws
IOException {
        List<File> schemas =
arquivosDaPastaEmOrdem.stream()
                .map(nomeArquivoSchema -> new
File(pastaRaiz, nomeArquivoSchema))
                .toList();

        if (schemas.isEmpty()) {
            System.out.println("Nenhum arquivo de
schema (.avsc, .proto) encontrado.");
            return "";
        }

        // se contém apenas 1 arquivo, não é necessário
realizar merge. Basta adicionar o conteúdo
        // dentro do schema
        if(schemas.size() == 1){
            ObjectMapper mapper = new ObjectMapper();
            ObjectNode rootNode =
mapper.createObjectNode();
            var schemaType =
SchemaType.selecionar(schemas);
            if(schemaType != SchemaType.AVRO)
rootNode.put("schemaType", schemaType.name());
            rootNode.put("schema", new
String(Files.readAllBytes(schemas.getFirst().toPath()))
);

            return mapper.writeValueAsString(rootNode);
        }

        // Criar um único parser para registrar os
```

```
schemas
        Schema.Parser parser = new Schema.Parser();

        // Primeiro, carregamos todos os schemas e os
registramos no parser
        for (File schemaFile : schemas) {
            parser.parse(new
String(Files.readAllBytes(schemaFile.toPath())));
        }

        //depois adquirimos o último parser, visto que
ele é o raiz, e o schema
        String chave = parser.getTypes().keySet()
                .stream().reduce((primeiro, segundo) ->
segundo).orElse(null);

        String schemaMergeadoJson =
parser.getTypes().get(chave).toString(true);

        if(comSchema){
            ObjectMapper mapper = new ObjectMapper();
            ObjectNode rootNode =
mapper.createObjectNode();
            var schemaType =
SchemaType.selecionar(schemas);
            if(schemaType != SchemaType.AVRO)
rootNode.put("schemaType", schemaType.name());
            rootNode.put("schema", schemaMergeadoJson);

            return mapper.writeValueAsString(rootNode);
        }
        return schemaMergeadoJson;
    }

}
```

Para manter a construção dos arquivos de cadastro de *schemas* Avro iguais, omitimos a inclusão do elemento `schemaType` quando os arquivos usam a extensão `.avsc`.

Os testes de `AvroSchemaMerger` sofreram algumas modificações. Confira:

```
package br.com.alura.avroschemamerger;

import org.junit.jupiter.api.Test;

import java.io.File;
import java.io.IOException;
import java.util.List;

import static java.nio.file.Files.readAllBytes;
import static
org.assertj.core.api.Assertions.assertThat;

class AvroSchemaMergerTest {

    private final File pastaRaiz = new
File("src/test/resources");
    private final List<String> nomesArquivos =
List.of("ItemCardapio.avsc", "ItemDoPedido.avsc",
"Endereco.avsc", "Pedido.avsc");
    private final String arquivoProtobuf =
"entregador.proto";
    private final AvroSchemaMerger merger = new
AvroSchemaMerger();
    private final File pastaResultado = new
File(pastaRaiz,"resultado/");
```

```
    @Test
    void mergeSimples() throws IOException {
        var merge = merger.merge(pastaRaiz, false,
nomesArquivos);

        var avscEsperado = new String(
                readAllBytes(new File(pastaResultado,
"Resultado.avsc").toPath()));

        assertThat(merge).isEqualTo(avscEsperado);
    }

    @Test
    void
mergeSimplesComSchemaProtobufParaEnvioViaRestParaSchema
Registry() throws IOException {
        var merge = merger.merge(pastaRaiz, true,
List.of(arquivoProtobuf));

        var schemaEsperado = new String(
                readAllBytes(new File(pastaResultado,
"entregadorcomschema.json").toPath()));

        assertThat(merge).isEqualTo(schemaEsperado);
    }

    @Test
    void
mergeComSchemaParaEnvioViaRestParaSchemaRegistry()
throws IOException {
        var merge = merger.merge(pastaRaiz, true,
nomesArquivos);

        var avscEsperado = new String(
                readAllBytes(new File(pastaResultado,
```

```
"ResultadoComSchema.avsc").toPath()));

        assertThat(merge).isEqualTo(avscEsperado);
    }
}
```

Com essas alterações, podemos criar um *script bash* que cadastre automaticamente o novo *schema*. Ele não deve ser muito diferente dos que já criamos:

```
cd ../avroschemamerger
./instalar_avroschemamerger.sh
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-notificacoes/src/main/protobuf true
entregador.proto > ../schemas/notificacoes-value.js

cd ../schemas

curl -X POST
http://localhost:8081/subjects/notificacoes-
value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@notificacoes-value.js
```

Seguindo o padrão de nomenclatura dos demais *scripts* dessa natureza, vamos chamá-lo de `notificacoes-value-cadastrar.sh`, salvá-lo na pasta `schemas` e referenciá-lo em `infra-up.sh`:

```
while true; do
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
```

```
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
    ./entregadores-value-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar-v2.sh
    ./pedidos-value-cadastrar-v2.sh
    ./entregadores-value-cadastrar-v2.sh
    ./notificacoes-value-cadastrar.sh
    break
  fi
  sleep 5
done
```

Ao subirmos os microsserviços:

- *ms-pedidos*
- *ms-entregadores*
- *ms-notificacoes*

E realizarmos um pedido usando o Postman ou o cURL a seguir:

```
curl --location 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o
limite",
        "preco": 50.00
      },
```

```
      "quantidade": 2
    }
  ],
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
'
```

O pedido será enviado aos entregadores e, uma vez que um deles for selecionado, uma notificação de entregador despachado é emitida pelo *ms-notificacoes*. A seguir, a notificação obtida do tópico `notificacoes`:

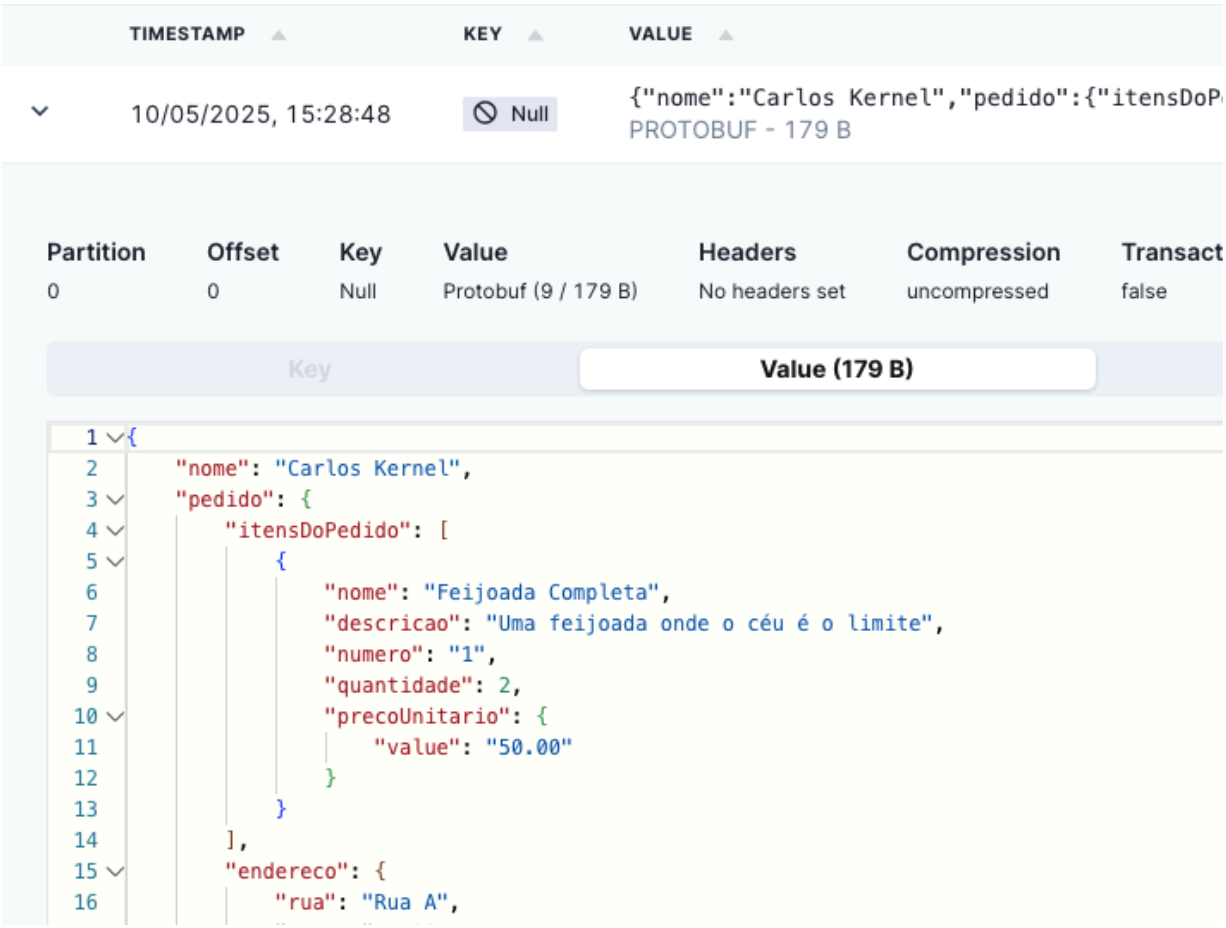


Figura 8.4: Notificação de entregador despachado, com um novo pedido.

8.7 KafkaProtobufDeserializer, o KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE e o consumo das notificações pelo ms-restaurantes

Com o envio das notificações, o *ms-restaurantes* poderá recebê-las e, assim, entender que os pedidos

foram despachados, bem como os entregadores responsáveis pelas entregas.

De forma semelhante ao que já fizemos em outros consumidores de mensagens Kafka, precisamos:

- inserir a biblioteca que contenha o desserializador apropriado e o *plugin* compilador de *schemas* Protobuf no `pom.xml`. No caso, usaremos o `kafka-protobuf-serializer` e o `protobuf-maven-plugin` (já usados no *ms-notificacoes*);
- obter o *schema* do tópico `notificacoes` no Schema Registry para inseri-lo na pasta `src/main/protobuf`. Para isso, podemos usar o cURL a seguir:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo codigo-fonte
% curl -X GET
http://localhost:8081/subjects/notificacoes-
value/versions/latest/schema
```

Ainda temos que:

- atualizar o `KafkaObjects` do *ms-restaurantes*, inserindo os *beans* de leitura do tópico;
- criar um receptor das notificações, bem como lógica relacionada ao tratamento delas.

Talvez a única mudança notável na construção dos *beans* do `KafkaObjects` esteja na necessidade de inserir a configuração

`KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE`, que indica a classe que deve ser usada para desserializar a mensagem. No caso, usaremos a `Entregador`, gerada pelo plugin `protobuf-maven-plugin`.

Adiante, o trecho de código necessário no `KafkaObjects`:

```
    @Bean

KafkaListenerContainerFactory<ConcurrentMessageListener
Container<Void, Entregador>> kafkaListener() {
        ConcurrentKafkaListenerContainerFactory<Void,
Entregador> factory =
                new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(
                consumerConfigs()));
        return factory;
    }

    Map<String, Object> consumerConfigs() {
        return Map.of(

ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaServerUrl
                ,
ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
VoidDeserializer.class
                ,
ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaProtobufDeserializer.class
                , ConsumerConfig.GROUP_ID_CONFIG, grupo
```

```
                ,
ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
autoOffsetReset
                ,
AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
IG, schemaRegistryUrl
                ,
KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE
_TYPE, Entregador.class.getName()
        );
    }
```

Já o receptor encontra a seguir:

```
@Component
@AllArgsConstructor
class Receptor {

    private PainelDeEntregadores painel;

    @KafkaListener(topics = "${topico.notificacoes}"
            , groupId = "${spring.kafka.consumer.group-
id}"
            , containerFactory = "kafkaListener")
    public void receber(Entregador entregadorComPedido)
{
        painel.exibir(entregadorComPedido);
    }
}
```

O `PainelDeEntregadores` é uma classe que representa um painel onde é possível visualizar os entregadores que foram despachados. Na sequência, confira como ele está implementado:

```
@Service
public class PainelDeEntregadores {

    public void exibir(Entregador entregadorComPedido)
{
        System.out.println("-----------------------");
        System.out.println("Entregador: " +
entregadorComPedido.getNome() + " enviado para entrega
de pedido");
        System.out.println("-----------------------");
    }
}
```

Com os microsserviços anteriores no ar, precisamos apenas subir o *ms-pedidos*. A cada pedido feito, um entregador é despachado e eles são exibidos no painel:

```
-----------------------
Entregador: Juliana Script enviado para entrega de
pedido
-----------------------
-----------------------
Entregador: Carlos Kernel enviado para entrega de
pedido
-----------------------
-----------------------
Entregador: Igor Pacote enviado para entrega de pedido
-----------------------
```

8.8 Resumo

Neste capítulo, vimos como enviar e receber mensagens no formato Protobuf. Usamos esse protocolo para enviar notificações de entregadores despachados para entrega de pedidos a todos os sistemas interessados.

No próximo capítulo, veremos uma outra ferramenta que, assim como o Schema Registry, é considerada satélite do Kafka: o Kafka Connect. E integraremos ambas as soluções.

CAPÍTULO 9

Integração com o Kafka Connect

Ao longo do livro, vimos que podemos usar diversos protocolos de comunicação para integrar nossas aplicações, cada um com suas vantagens e desvantagens.

Neste capítulo, entretanto, veremos um novo tipo de produtor de mensagens, o Kafka Connect. Com ele, até mesmo aplicações criadas com tecnologias que não suportam a criação de produtores ou consumidores de tópicos Kafka poderão interagir com essa ferramenta.

9.1 Consulta aos pedidos realizados

No contexto do Alura Eats, é fundamental que diversos sistemas consigam acessar, em tempo real, os pedidos realizados pelos clientes. Esses dados são importantes tanto para o processamento logístico da entrega quanto para múltiplos propósitos estratégicos, como geração de relatórios, análises preditivas, sistemas de recomendação e auditoria.

Inicialmente pensamos em criar, a partir do *ms-pedidos*, um outro produtor de mensagens Kafka (além do `PedidosSenderKafka`), que enviaria os pedidos realizados para um tópico Kafka adequado. Entretanto, optou-se por não inserir essa funcionalidade no sistema, visto que ela não faz parte do contexto delimitado de regras de negócio a que o *ms-pedidos* deve atender.

Um contexto delimitado (em inglês, *bounded context*) é um conceito relacionado ao *Domain Driven Design* (DDD) e representa um limite explícito dentro do qual um modelo possui um significado claro e consistente, sem ambiguidades nem mistura de conceitos relacionados a outros contextos.

Na nossa solução, o *ms-pedidos* foi criado para lidar especialmente com o cliente, que utiliza o sistema para fazer suas compras. Para tanto, ele oferece os serviços de consulta ao cardápio dos restaurantes e lanchonetes, além de compra efetiva. Esse, portanto, é o seu contexto. A funcionalidade de consulta aos pedidos realizados não faz parte das funcionalidades que um cliente precisa (ou mesmo deseja) acessar.

Para mais informações sobre esse conceito, as seguintes leituras são recomendadas:

- *Domain Driven Design: Atacando as Complexidades no Coração do Software*, de Eric Evans (2016)
- *Implementando Domain Driven Design*, de Vaughn Vernon (2016)

Para que os pedidos realizados fossem enviados ao tópico destinado à auditoria, consideramos essencial adotar uma solução que atuasse de forma transparente para o *ms-pedidos* — ou seja, sem exigir alterações em seu código. Nesse contexto, o Kafka Connect mostrou-se a solução ideal.

9.2 O Kafka Connect

Assim como o Schema Registry, o Kafka Connect é uma ferramenta auxiliar ao Kafka, projetada para integrar o Kafka com sistemas externos. Ele permite movimentar dados entre o Kafka e outras fontes de dados (sejam elas bancos de dados, filas, serviços em nuvem e outras) sem que seja necessária a escrita de código manual para isso. Assim, ele conta com um conjunto de *connector plugins* criados por diversos fabricantes. Esses *connector plugins*:

- Podem enviar dados de um banco para o tópico Kafka. São chamados de conectores *source*.
- Podem enviar dados de um tópico Kafka para uma fonte de dados. São chamados de conectores *sink*.

Uma vez que temos o *connector plugin* adequado, basta que o configuremos usando uma configuração

em formato JSON, onde inserimos diversas configurações, entre elas:

- a origem dos dados;
- o destino dos dados;
- e alguns ajustes finos.

O Kafka Connect, nesse ponto, possui certa semelhança com o acesso aos bancos de dados usando JDBC:

- o *connector plugin* se assemelha ao driver JDBC;
- a configuração JSON é parecida com a String de conexão com o banco, exceto que essa última possui mais informações que precisam ser configuradas.

> Para mais informações sobre o Kafka Connect, uma bibliografia recomendada é a *Kafka Connect e Kafka Streams: Integre suas aplicações ao Kafka usando as melhores práticas* (2024).

Veremos a seguir como utilizamos o Kafka Connect na nossa solução, integrada ao Schema Registry.

9.3 Envio dos pedidos realizados para o tópico de auditoria usando o Kafka Connect

Basicamente, uma vez que os pedidos são realizados, eles são persistidos na *collection* `pedidos` presente no MongoDB da nossa solução. Assim que ocorre a persistência dos dados no banco, o Kafka Connect os envia como uma mensagem para o tópico `mongo.restaurantes.pedidos`. No entanto, antes de enviá-la, ela passa pelo Schema Registry para verificar se está conforme o esperado para o tópico.

A figura a seguir mostra a arquitetura dessa solução.

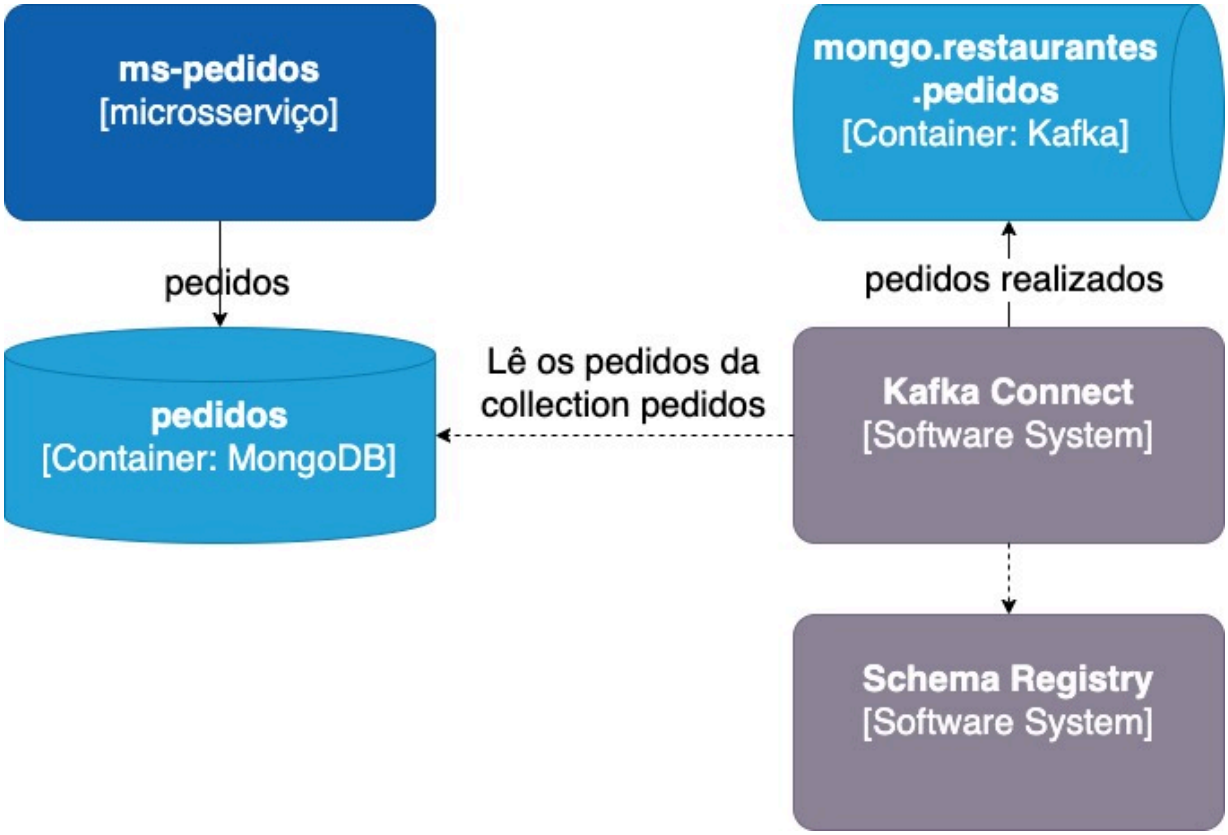


Figura 9.1: Arquitetura de envio de pedidos realizados para auditoria.

Podemos considerar o Kafka Connect como um produtor de mensagens. Como fizemos em todos os demais produtores, usaremos o Schema Registry para validar as mensagens enviadas.

Para que isso seja possível, inicialmente precisamos configurar o Kafka Connect no nosso `docker-compose.yml` juntamente ao *connector source* apropriado para acessar o MongoDB. Usaremos o conector da Debezium, que é de código aberto e recomendado pela Confluent no seu catálogo de conectores.

> A lista com todos os conectores recomendados pela Confluent encontra-se em https://www.confluent.io/hub/

O *connector plugin* é um conjunto de arquivos `jar` que deve ser inserido em pastas específicas do Kafka Connect. Assim que a ferramenta sobe, ela varre esses diretórios em busca dos arquivos, e os carrega em *classloaders* separados — um para cada *connector plugin*. Dessa forma, em caso de problemas com algum deles, os demais continuam funcionando normalmente.

Para configurar o Kafka Connect no `docker-compose.yml` temos algumas alternativas:

- inserir o Kafka Connect com todas as configurações necessárias e criar um volume que contenha os *connector plugins* e outros arquivos necessários;
- criar uma imagem própria, contendo o Kafka Connect e os *connector plugins*.

Usamos a segunda opção. Assim, não será necessário armazenar os *connector plugins* no nosso projeto, mas baixá-los quando for necessário. Para tanto, criamos um `Dockerfile` com uma imagem do Kafka Connect como imagem base. Esse `Dockerfile` também contém a pasta onde os *connector plugins* serão armazenados e cria uma variável de ambiente com o caminho para essa pasta. Além disso, inclui instruções para que o *connector plugin* da Debezium seja baixado, descompactado e seus arquivos enviados para a pasta correta. Por fim, o arquivo contém instruções para instalação de arquivos que permitem que o Debezium suporte o Schema Registry da Confluent.

A seguir, o arquivo com todas essas instruções:

```
FROM confluentinc/cp-kafka-connect-base:7.9.1

ENV KAFKA_CONNECT_PLUGINS_DIR=/etc/kafka-connect/jars
RUN mkdir -p $KAFKA_CONNECT_PLUGINS_DIR

RUN echo ">>>>>>>>>> Instalando Plugin do mongodb
```

```
<<<<<<<<<<"
RUN wget
https://repo1.maven.org/maven2/io/debezium/debezium-
connector-mongodb/3.1.1.Final/debezium-connector-
mongodb-3.1.1.Final-plugin.tar.gz
RUN tar xvfz debezium-connector-mongodb-3.1.1.Final-
plugin.tar.gz --directory $KAFKA_CONNECT_PLUGINS_DIR

RUN echo ">>>>>>>>>> Instalando suporte ao Schema
Registry da Confluent <<<<<<<<<<"
RUN confluent-hub install --no-prompt --component-dir
$KAFKA_CONNECT_PLUGINS_DIR confluentinc/kafka-connect-
avro-converter:7.9.1
```

Conforme a documentação da Debezium, presente em https://debezium.io/documentation/reference/3.1/configuration/avro.html, os conectores Debezium não oferecem mais suporte ao Schema Registry da Confluent desde a versão 2.0.0, e orientam a instalação de algumas bibliotecas juntamente dos demais *connector plugins* para que a integração seja possível com os *connector plugins* fabricados por eles.

Além disso, o comando `confluent-hub` é um utilitário presente em diversas imagens docker oferecidas pela Confluent e permite um acesso conveniente a diversas funcionalidades que a empresa oferece, como download de dependências e instalação de *connector plugins*. Para mais informações, acesse https://docs.confluent.io/platform/7.5/connect/confluent-hub/client.html

Para que o Kafka Connect entenda que deve fazer uma varredura na pasta descrita no `Dockerfile`, precisamos informar isso no `docker-compose.yml`. Além disso, precisamos declarar:

- o endereço onde está o Kafka que ele deve acessar (por meio da variável

`CONNECT_BOOTSTRAP_SERVERS`).

- a porta pela qual podemos acessar o Kafka Connect utilizando o protocolo HTTP. Útil para a criação de conectores, conforme veremos mais adiante. Usamos a variável `CONNECT_REST_PORT` com esse objetivo.
- as variáveis `CONNECT_LISTENERS` e `CONNECT_REST_ADVERTISED_HOST_NAME` são usadas para fornecer um endereço HTTP interno para as aplicações. Usaremos essas configurações para conectarmos o Redpanda Console ao Kafka Connect mais tarde.
- os tópicos de controle da aplicação. O Kafka Connect não utiliza banco de dados para armazenar suas informações dinâmicas (como conectores criados, offsets de mensagens etc.), mas tópicos Kafka, e para essa finalidade usa as variáveis `CONNECT_CONFIG_STORAGE_TOPIC`, `CONNECT_OFFSET_STORAGE_TOPIC` e `CONNECT_STATUS_STORAGE_TOPIC`.
- os conversores padrão da ferramenta. Semelhante aos serializadores que usamos em produtores e consumidores Kafka, eles convertem os registros lidos das fontes de dados para um formato específico. No nosso caso, configuramos ambos os conversores de chave e valor das mensagens para o formato `AvroConverter`.

Usamos, para tal fim, as variáveis `CONNECT_KEY_CONVERTER` e `CONNECT_VALUE_CONVERTER`.

- o Schema Registry que será utilizado pela ferramenta na validação das chaves e dos valores das mensagens. Usaremos o mesmo Schema Registry em ambos os casos, e as variáveis a serem utilizadas são, respectivamente, `CONNECT_KEY_CONVERTER_SCHEMA_REGISTRY_URL` e `CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL`.
- as pastas onde se encontram os *connector plugins*. Pode-se usar mais de uma pasta, desde que todas sejam declaradas. Usamos a variável `CONNECT_PLUGIN_PATH` para armazenar essa informação.

O trecho do `docker-compose.yml` contendo as configurações para uso do Kafka Connect está a seguir:

```
kafka-connect:
  build:
    context: ./kafka-connect
    dockerfile: Dockerfile
  hostname: kafka-connect
  container_name: kafka-connect
  restart: always
  depends_on:
    - kafka
    - mongo
    - schema-registry
  ports:
```

```
        - "9083:8083"
      environment:
          CONNECT_BOOTSTRAP_SERVERS: kafka:29092
          CONNECT_REST_PORT: 9083
          CONNECT_LISTENERS: HTTP://0.0.0.0:8083
          CONNECT_REST_ADVERTISED_HOST_NAME: kafka-
connect
          CONNECT_GROUP_ID: "connect-cluster"
          CONNECT_CONFIG_STORAGE_TOPIC: "_connect-
configs"
          CONNECT_OFFSET_STORAGE_TOPIC: "_connect-
offsets"
          CONNECT_STATUS_STORAGE_TOPIC: "_connect-
status"
          CONNECT_KEY_CONVERTER:
"io.confluent.connect.avro.AvroConverter"
          CONNECT_VALUE_CONVERTER:
"io.confluent.connect.avro.AvroConverter"
          CONNECT_KEY_CONVERTER_SCHEMA_REGISTRY_URL:
"http://schema-registry:8081"
          CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL:
"http://schema-registry:8081"
          CONNECT_PLUGIN_PATH:
"/usr/share/java,/etc/kafka-connect/jars"
      networks:
          bubble:
```

Com o Kafka Connect devidamente instalado com o *connector plugin* para acesso ao MongoDB, precisamos configurar um conector, que fará a ligação entre a *collection* `pedidos` e o tópico `mongo.restaurantes.pedidos`. Para isso, enviamos um JSON via POST para o Kafka Connect, contendo as seguintes informações:

- o nome do conector, expresso pela configuração `name`.
- o `connector.class`, que contém a classe que será usada como conector. Ela deve estar no *classpath* do Kafka Connect. Como já inserimos o *connector plugin* na ferramenta, isso não será um problema.
- a URL de acesso ao banco de dados, informação presente em `mongodb.connection.string`.
- o nome do banco de dados, expresso em `mongodb.name`.
- os bancos de dados que contêm *collections* a serem lidas pelo conector. Use `database.include.list` para armazenar essa informação.
- as *collections* que serão lidas pelo conector, com a configuração `collection.include.list`.
- um prefixo para o nome do tópico; neste caso, foi usado no conector "mongo", pois o nome do tópico é `mongo.restaurantes.pedidos`. Por padrão, o Kafka Connect envia as mensagens para um tópico cujo nome é a junção de algumas informações:
  - um prefixo, que pode ser definido livremente;
  - o nome do banco de dados que será lido;
  - o nome da *collection* que será lida.
- os conversores de chave e valor das mensagens, embora já declarados no `docker-compose.yml`,

também são obrigatórios nos conectores. Podemos utilizar os já registrados no Kafka Connect ou sobrescrevê-los, se necessário. Usamos, portanto, o `StringConverter` na conversão de chaves — apenas como demonstração de sobrescrita — e o `AvroConverter` na conversão dos valores, inserindo-os nas configurações `key.converter` e `value.converter`.

- o endereço do Schema Registry a ser usado pelo conector, na configuração `value.converter.schema.registry.url`.
- a configuração `value.converter.auto.register.schemas`, que equivale à variável `AUTO_REGISTER_SCHEMAS`, usada nos produtores de mensagens em capítulos anteriores.

Como podemos ver, o Kafka Connect nesse caso será um produtor de mensagens. Logo, precisamos de configurações iguais ou muito semelhantes às usadas nos demais produtores, acrescidas de outras exclusivas para a ferramenta.

Observe que configuramos o Schema Registry para ser usado apenas no valor da mensagem. Mas poderíamos também configurá-lo para a chave da mensagem. Nesse caso, usaríamos as configurações análogas às de `value.converter`: as de `key.converter`. Assim, teríamos configurações como `key.converter.schema.registry.url`, `key.converter.schemas.enabled` e outras.

Configurações avançadas do Kafka Connect

- `value.converter.schemas.enabled`, que informa que o *schema* da mensagem não precisa ser enviado junto com a mesma para o Kafka. O Kafka Connect faz isso por padrão, para que os consumidores possuam informações sobre a estrutura da mensagem. Entretanto, como usaremos o Schema Registry para armazenar o *schema*, essa operação torna-se desnecessária. Vamos configurá-la com `false`.

- o Kafka Connect permite que as mensagens sejam transformadas antes de serem enviadas para o tópico, se necessário. Para isso, utilizamos o conceito de *Single Message Transformations* (SMTs). Com isso, podemos realizar ajustes como copiar ou mover informações presentes no corpo da mensagem para o *header* ou para a chave, selecionar mensagens a serem enviadas segundo algum critério, realizar roteamento de mensagens para tópicos de acordo com alguma informação da mensagem, entre outras ações. No nosso caso, usamos o `io.debezium.connector.mongodb.transforms.ExtractNewDocumentState`, que obtém apenas o conteúdo da mensagem que nos interessa, que são os campos vindos da *collection* `pedidos`.

Rotear mensagens para tópicos ou filas baseadas no seu conteúdo é, inclusive, um padrão catalogado no livro *Enterprise Integration Patterns* (2003) de Gregor Hohpe, como Context-Based Router. Tanto a Confluent quanto a Debezium e outros fabricantes já desenvolveram diversos SMTs. O livro *Kafka Connect e Kafka Streams - Integre suas aplicações ao Kafka usando as melhores práticas* (2024) utilizou diversos deles. Vale a pena dar uma olhada.

Uma mensagem produzida pelo Kafka Connect contém diversos campos. Entre eles, informações como:

- a versão do conector;
- o registro no banco "antes" (`before`) e "depois" (`after`) da manipulação de dados. Como lidamos apenas com inserções no banco, o trecho da mensagem que contém o registro "antes" é vazio, e o trecho do "depois" inclui os campos inseridos no banco.

Adiante, um esboço da mensagem produzida pelo Kafka Connect, caso nenhum SMT tenha sido inserido:

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"valor da coluna value ou campo de
mesmo nome, no caso de collection mongodb"
    },
    "source": {
        ...
        "snapshot":"incremental"
    },
    "op":"r",
    "ts_ms":"1620404023235",
    "ts_us":"16204040232253443",
    "ts_ns":"162040402322542232247",
```

```
    "transaction":null
}
```

Basicamente, o `ExtractNewDocumentState` obtém apenas as informações do elemento `after`. Dessa forma, a mensagem no tópico Kafka torna-se consideravelmente menor e mais simples de ser manipulada.

O conector que usaremos está a seguir, localizado no arquivo `pedidos-realizados.sh` presente na pasta `docker/kafka-connect`:

```
curl -X POST http://localhost:9083/connectors \
  -H "Content-Type: application/json" \
  -d '{
    "name": "pedidos-realizados-connector",
    "config": {
      "connector.class":
"io.debezium.connector.mongodb.MongoDbConnector",
      "mongodb.connection.string":
"mongodb://user:password@mongo:27017/?replicaSet=rs0",
      "mongodb.name": "restaurantes",
      "database.include.list": "restaurantes",
      "collection.include.list":
"restaurantes.pedidos",

      "topic.prefix": "mongo",

      "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
      "value.converter":
"io.confluent.connect.avro.AvroConverter",
      "value.converter.schema.registry.url":
```

```
"http://schema-registry:8081",
      "value.converter.auto.register.schemas": "true",
      "value.converter.schemas.enabled": "false",

      "skipped.operations": "u,d",

      "transforms": "unwrap",
      "transforms.unwrap.type":
"io.debezium.connector.mongodb.transforms.ExtractNewDoc
umentState",
      "transforms.unwrap.drop.tombstones": "true"
    }
  }'
```

> Repare que inserimos o valor `true` na propriedade `value.converter.auto.register.schemas` no conector. Ainda neste capítulo veremos por que isso é necessário.

Com o *script* criado, vamos referenciá-lo no final de `infra-up.sh`, para que ele seja cadastrado no Kafka Connect. Como temos diversos contêineres rodando, pode ser que o Kafka Connect demore um pouco para subir. De forma semelhante ao que fizemos com o Schema Registry, antes de cadastrar o conector, vamos verificar se o Kafka Connect já subiu:

```
echo aguardando o Kafka Connect subir para cadastrar os
conectores
while true; do
  HTTP_STATUS=$(curl -s -o /dev/null -w "%{http_code}"
```

```
$CONNECT_URL)

  if [ "$HTTP_STATUS" -eq 200 ]; then
    echo "Kafka Connect está pronto para cadastro de
conectores!"
    echo "Executando scripts:"
    ./pedidos-realizados.sh
    break
  fi

  sleep 5
done
```

Também podemos configurar o Redpanda Console para que possamos usá-lo como console do Kafka Connect. Para isso, complementamos o seu arquivo de configuração (`CONSOLE_CONFIG_FILE`) com as seguintes informações:

- `connect` , para indicar que será configurado um Kafka Connect;
- informar que ele está habilitado com a configuração `enabled` ;
- e mencionar os `clusters` da ferramenta.

A configuração completa do Redpanda Console com essas alterações encontra-se a seguir:

```
    redpanda:
      image:
docker.redpanda.com/redpandadata/console:v2.8.2
      container_name: redpanda
      ports:
```

```
      - 18080:8080
    depends_on:
      - kafka
      - kafka-connect
    restart: always
    entrypoint: /bin/sh
    command: -c "echo \"$$CONSOLE_CONFIG_FILE\" >
/tmp/config.yml; /app/console"
    environment:
      CONFIG_FILEPATH: /tmp/config.yml
      CONSOLE_CONFIG_FILE: |
        kafka:
          brokers: ["kafka:29092"]
          schemaRegistry:
            enabled: true
            urls: ["http://schema-registry:8081"]
          protobuf:
            enabled: true
            schemaRegistry:
              enabled: true
              refreshInterval: 5s
        connect:
          enabled: true
          clusters:
            - name: kafka-connect
              url: http://kafka-connect:8083
networks:
    bubble:
```

Resumindo:

- criamos uma instância de Kafka Connect, com o *connector plugin* adequado à fonte de dados que estamos usando;

- configuramos um conector que lê os dados da *collection* `pedidos` e os envia para o tópico `mongo.restaurantes.pedidos`;
- atualizamos o Redpanda Console para que ele seja não somente a interface do Kafka e do Schema Registry, mas também do Kafka Connect.

Ao subirmos a infraestrutura da solução, podemos acessar o Redpanda Console e clicar em "Connect" no menu à esquerda. O *cluster* de Kafka Connect e o conector cadastrado devem aparecer, conforme a figura:

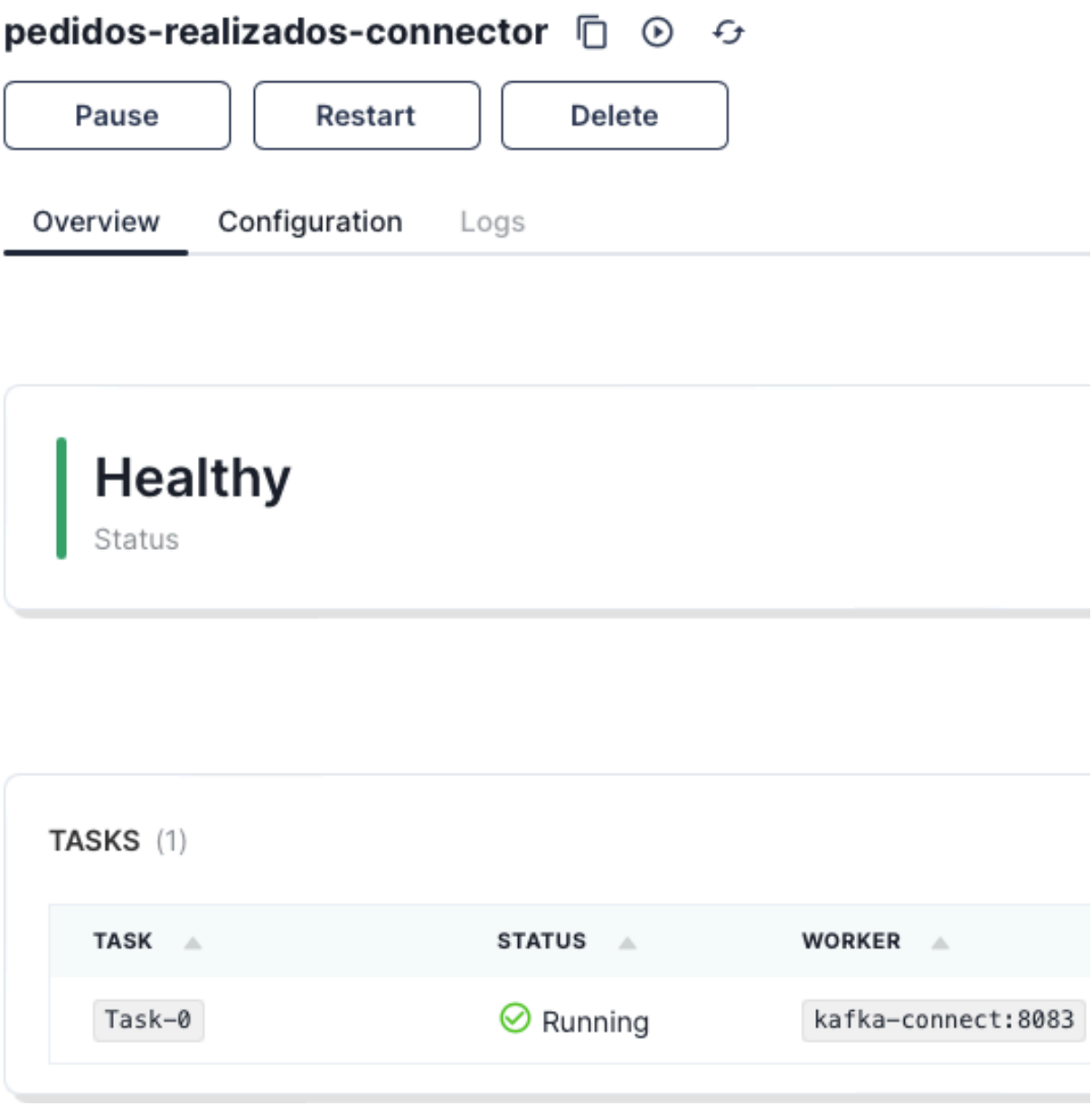


Figura 9.2: Redpanda Console exibindo informações sobre o Kafka Connect e o conector cadastrado.

Para testar o envio de mensagens para o tópico `mongo.restaurantes.pedidos`, precisamos também subir o *ms-pedidos*:

```
mvn -f ms-pedidos/pom.xml spring-boot:start
```

E fazer um pedido:

```
curl --location 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o
limite",
        "preco": 50.00
      },
      "quantidade": 2
    }
  ],
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
'
```

Note que, embora diversas alterações fossem necessárias neste capítulo, nenhuma linha de código precisou ser modificada no *ms-pedidos*. Isso é possível pois o Kafka Connect não se conecta ao microsserviço, mas ao seu banco de dados. Na prática, o *ms-pedidos* não sabe da existência do Kafka Connect na solução, e nem precisa saber.

O *ms-pedidos* envia o pedido para o banco de dados, e o Kafka Connect, por sua vez, envia o registro para o tópico:

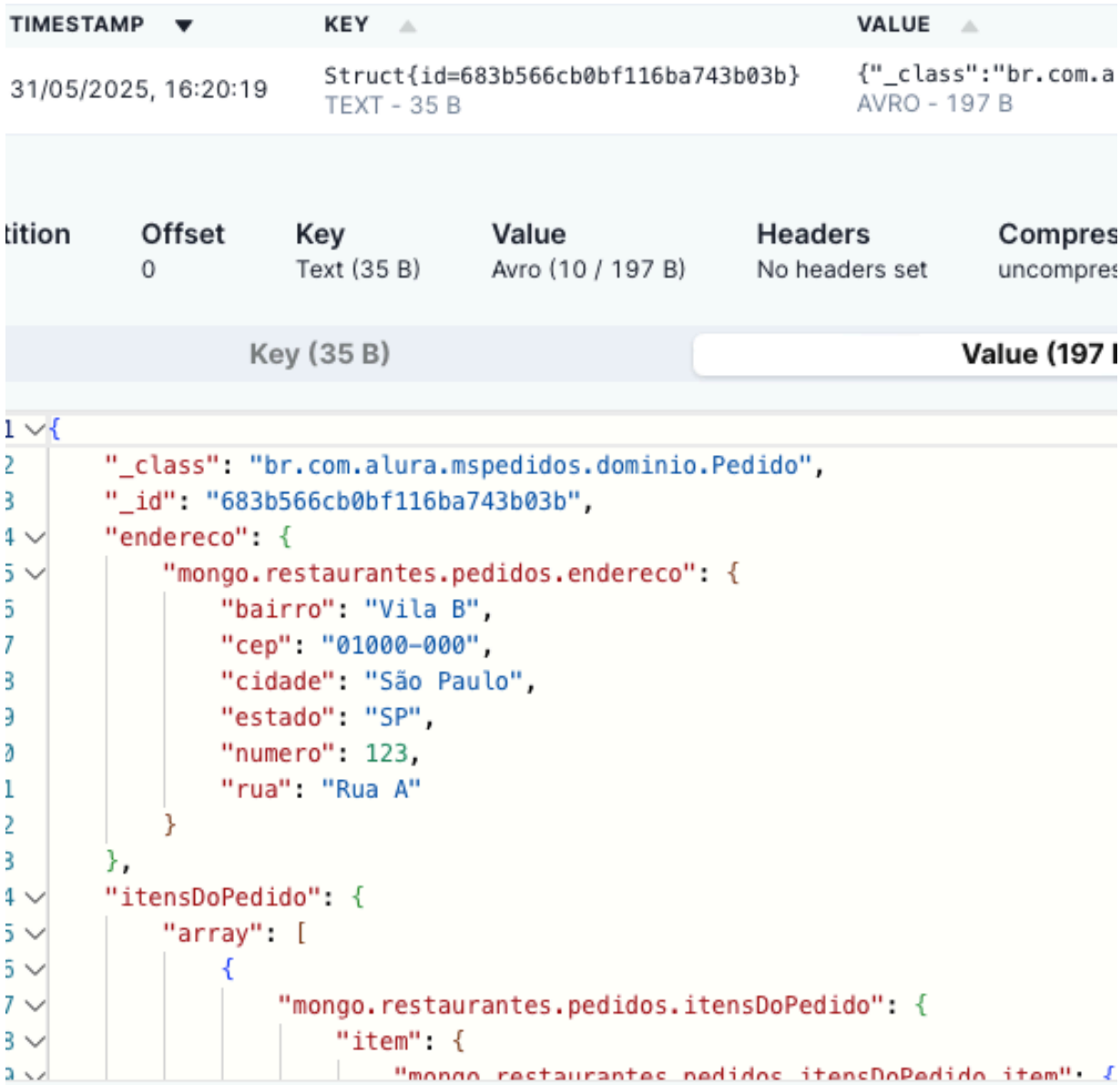


Figura 9.3: Primeiro pedido realizado, enviado para o Kafka pelo Kafka Connect.

Podemos verificar que, graças à configuração `value.converter.auto.register.schemas`, foi inserido um *schema* no Schema Registry:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl
-X GET
```

http://localhost:8081/subjects/mongo.restaurantes.pedidos-value/versions/1/schema


```
{"type":"record","name":"pedidos","namespace":"mongo.restaurantes","fields":[{"name":"_id","type":
["null","string"],"default":null},
{"name":"itensDoPedido","type":["null",
{"type":"array","items":["null",
{"type":"record","name":"itensDoPedido","namespace":"mongo.restaurantes.pedidos","fields":
[{"name":"item","type":["null",
{"type":"record","name":"item","namespace":"mongo.restaurantes.pedidos.itensDoPedido","fields":
[{"name":"numero","type":
["null","long"],"default":null},{"name":"nome","type":
["null","string"],"default":null},
{"name":"descricao","type":
["null","string"],"default":null},
{"name":"preco","type":
["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido.item"}],"default":
null},{"name":"quantidade","type":
["null","int"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido"}]}],"default":null},
{"name":"endereco","type":["null",
{"type":"record","name":"endereco","namespace":"mongo.restaurantes.pedidos","fields":[{"name":"rua","type":
["null","string"],"default":null},
{"name":"numero","type":["null","int"],"default":null},
{"name":"bairro","type":
["null","string"],"default":null},
{"name":"cidade","type":
["null","string"],"default":null},
{"name":"estado","type":
["null","string"],"default":null},{"name":"cep","type":
["null","string"],"default":null},
{"name":"complemento","type":
```

```
["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.endereco"}],"default":null},
{"name":"total","type":
["null","string"],"default":null},
{"name":"_class","type":
["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos"}
```


No *schema* mostrado é possível notar que:

- todos os campos não possuem obrigatoriedade de preenchimento;
- está faltando o campo `complemento`.

Vamos então enviar um outro pedido, agora com um endereço com complemento:

```
curl --location 'http://localhost:8082/pedidos' \
--header 'Content-Type: application/json' \
--data '{
  "itens": [
    {
      "item": {
        "numero": 1,
        "nome": "Feijoada Completa",
        "descricao": "Uma feijoada onde o céu é o limite",
        "preco": 50.00
      },
      "quantidade": 3
    }
  ],
  "endereco": {
    "rua": "Rua B",
    "numero": 456,
```

```
    "bairro": "Vila B",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-001",
    "complemento": "Bloco C, Ap. 456"
  }
}
'
```

Esse segundo pedido também é exibido no tópico em poucos instantes:

Figura 9.4: Segundo pedido realizado.

Mas, se olharmos no Schema Registry, mais um *schema* foi criado, agora contendo o campo `complemento`.

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl -X GET http://localhost:8081/subjects/mongo.restaurantes.pedidos-value/versions/2/schema
{"type":"record","name":"pedidos","namespace":"mongo.restaurantes","fields":[{"name":"_id","type":["null","string"],"default":null},{"name":"itensDoPedido","type":["null",{"type":"array","items":["null",{"type":"record","name":"itensDoPedido","namespace":"mongo.restaurantes.pedidos","fields":[{"name":"item","type":["null",{"type":"record","name":"item","namespace":"mongo.restaurantes.pedidos.itensDoPedido","fields":[{"name":"numero","type":["null","long"],"default":null},{"name":"nome","type":["null","string"],"default":null},{"name":"descricao","type":["null","string"],"default":null},{"name":"preco","type":["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido.item"}],"default":null},{"name":"quantidade","type":["null","int"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido"}]}],"default":null},{"name":"endereco","type":["null",{"type":"record","name":"endereco","namespace":"mongo.restaurantes.pedidos","fields":[{"name":"rua","type":["null","string"],"default":null},{"name":"numero","type":["null","int"],"default":null},{"name":"bairro","type":
```

```
["null","string"],"default":null},
{"name":"cidade","type":
["null","string"],"default":null},
{"name":"estado","type":
["null","string"],"default":null},{"name":"cep","type":
["null","string"],"default":null},
{"name":"complemento","type":
["null","string"],"default":null}],"connect.name":"mong
o.restaurantes.pedidos.endereco"}],"default":null},
{"name":"total","type":
["null","string"],"default":null},
{"name":"_class","type":
["null","string"],"default":null}],"connect.name":"mong
o.restaurantes.pedidos"}
```


Isso acontece devido ao funcionamento do Kafka Connect, que veremos a seguir.

9.4 Kafka Connect e Schema Registry: uma relação baseada em compatibilidade

Quando o Kafka Connect registra automaticamente um *schema* no Schema Registry, ele precisa garantir que o *schema* seja compatível com possíveis variações futuras dos dados.

Como os sistemas externos (como bancos de dados) podem ter colunas nulas, ausentes ou inconsistentes, o Kafka Connect adota uma estratégia defensiva:

cadastrar todos os campos de um *schema* como opcionais. Dessa forma, mensagens com dados parciais (contendo apenas alguns campos preenchidos) ainda seriam aceitas e compatíveis com o *schema*, evitando falhas no envio.

Esse comportamento é definido pela forma como o `AvroConverter` (usado no nosso conector) converte a estrutura interna dos dados do Kafka Connect (do tipo `Struct` ou `Map`) para um *schema* Avro: por não saber se determinado campo estará sempre presente, o Kafka Connect assume que nenhum campo é obrigatório.

Além disso, o `AvroConverter` ainda utiliza um *schema* com campos opcionais ao serializar os dados internamente. Isso significa que:

- a mensagem carrega um *schema*, e esse deve ser comparado com o registrado no Schema Registry;
- se os *schemas* possuírem o mesmo formato, o Schema Registry aprova a mensagem.

É por esse motivo que criamos o conector com a configuração `value.converter.auto.register.schemas` como `true` . Agora temos todas as possibilidades de *schemas* a serem utilizados pela nossa solução. Um *schema* com o campo `complemento` e outro sem.

Para desenvolvermos nossa solução, essa estratégia pode ser uma boa ideia. Contudo, em ambientes de produção, isso não é recomendável, pois trabalhar dessa maneira fará com que mensagens em diversos formatos sejam entregues para o Kafka, e os consumidores podem não estar preparados para tantos formatos diferentes. Então, podemos fazer o seguinte:

- criar 2 arquivos de *schemas*, com os *schemas* registrados no Schema Registry pelo Kafka Connect;
- cadastrar esses *schemas* via *scripts* adequados, assim como fizemos com todos os demais *schemas*;
- alterar a configuração `value.converter.auto.register.schemas` para `false`.

Dessa forma, os *schemas* que funcionaram com o Kafka Connect nos nossos testes passarão pelo mesmo processo de cadastro de *schemas* pelo qual todos os *schemas* passaram. Além disso, garantimos que mais nenhum novo *schema* seja registrado pelo Kafka Connect.

Vamos criar os arquivos `pedidos-realizados.avsc` e `pedidos-realizados-v2.avsc`. O conteúdo deles é bem semelhante. Será exibido adiante apenas o conteúdo de `pedidos-realizados.avsc`:

```
{
  "type": "record",
  "name": "pedidos",
  "namespace": "mongo.restaurantes",
  "fields": [
    {
      "name": "_id",
      "type": [
        "null",
        "string"
      ],
      "default": null
    },
    {
      "name": "itensDoPedido",
      "type": [
        "null",
        {
          "type": "array",
          "items": [
            "null",
            {
              "type": "record",
              "name": "itensDoPedido",
              "namespace":
"mongo.restaurantes.pedidos",
              "fields": [
                {
                  "name": "item",
                  "type": [
                    "null",
                    {
                      "type": "record",
                      "name": "item",
                      "namespace":
"mongo.restaurantes.pedidos.itensDoPedido",
```

```
                "fields": [
                  {
                    "name": "numero",
                    "type": [
                      "null",
                      "long"
                    ],
                    "default": null
                  },
                  {
                    "name": "nome",
                    "type": [
                      "null",
                      "string"
                    ],
                    "default": null
                  },
                  {
                    "name": "descricao",
                    "type": [
                      "null",
                      "string"
                    ],
                    "default": null
                  },
                  {
                    "name": "preco",
                    "type": [
                      "null",
                      "string"
                    ],
                    "default": null
                  }
                ],
                "connect.name":
"mongo.restaurantes.pedidos.itensDoPedido.item"
```

```
                    }
                  ],
                  "default": null
                },
                {
                  "name": "quantidade",
                  "type": [
                    "null",
                    "int"
                  ],
                  "default": null
                }
              ],
              "connect.name":
"mongo.restaurantes.pedidos.itensDoPedido"
            }
          ]
        }
      ],
      "default": null
    },
    {
      "name": "endereco",
      "type": [
        "null",
        {
          "type": "record",
          "name": "endereco",
          "namespace": "mongo.restaurantes.pedidos",
          "fields": [
            {
              "name": "rua",
              "type": [
                "null",
                "string"
              ],
```

```
    "default": null
  },
  {
    "name": "numero",
    "type": [
      "null",
      "int"
    ],
    "default": null
  },
  {
    "name": "bairro",
    "type": [
      "null",
      "string"
    ],
    "default": null
  },
  {
    "name": "cidade",
    "type": [
      "null",
      "string"
    ],
    "default": null
  },
  {
    "name": "estado",
    "type": [
      "null",
      "string"
    ],
    "default": null
  },
  {
    "name": "cep",
```

```
              "type": [
                "null",
                "string"
              ],
              "default": null
            },
            {
              "name": "complemento",
              "type": [
                "null",
                "string"
              ],
              "default": null
            }
          ],
          "connect.name":
"mongo.restaurantes.pedidos.endereco"
        }
      ],
      "default": null
    },
    {
      "name": "total",
      "type": [
        "null",
        "string"
      ],
      "default": null
    },
    {
      "name": "_class",
      "type": [
        "null",
        "string"
      ],
      "default": null
```

```
    }
  ],
  "connect.name": "mongo.restaurantes.pedidos"
}
```

Para facilitar a integração, vamos criar apenas um *script bash* para cadastrar os 2 *schemas*. Chamado de `pedidos-realizados-cadastrar.sh` , ele está localizado na pasta `schemas` :

```
cd ../avroschemamerger
./instalar_avroschemamerger.sh
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-pedidos/src/main/connect true pedidos-
realizados.avsc > ../schemas/pedidos-realizados.js
java -jar target/avro-schema-merger-0.0.1-SNAPSHOT.jar
../ms-pedidos/src/main/connect true pedidos-realizados-
v2.avsc > ../schemas/pedidos-realizados-v2.js

cd ../schemas

curl -X POST
http://localhost:8081/subjects/mongo.restaurantes.pedid
os-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@pedidos-realizados.js

curl -X POST
http://localhost:8081/subjects/mongo.restaurantes.pedid
os-value/versions \
 -H 'Content-Type:
application/vnd.schemaregistry.v1+json' \
 -d@pedidos-realizados-v2.js
```

O *script bash* deve ser inserido junto dos demais *scripts* em `infra-up.sh`:

```
while true; do
  if curl -s --head "$SCHEMA_REGISTRY_URL/config" |
head -n 1 | grep "200 OK" > /dev/null; then
    ./restaurantes.cardapio.itens-key-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar.sh
    ./pedidos-value-cadastrar.sh
    ./entregadores-value-cadastrar.sh
    ./restaurantes.cardapio.itens-value-cadastrar-v2.sh
    ./pedidos-value-cadastrar-v2.sh
    ./entregadores-value-cadastrar-v2.sh
    ./notificacoes-value-cadastrar.sh
    ./pedidos-realizados-cadastrar.sh
    break
  fi
  sleep 5
done
```

Por último, vamos alterar a configuração `value.converter.auto.register.schemas`, presente no *script* `pedidos-realizados.sh`:

```
# demais códigos omitidos por brevidade

"value.converter.schema.registry.url": "http://schema-
registry:8081",
"value.converter.auto.register.schemas": "false",
"value.converter.schemas.enabled": "false",
```

e cadastrar novamente o conector:

```
rodrigovieirapinto@MacBook-Pro-de-Rodrigo codigo-fonte
% cd docker
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker % curl
-X DELETE http://localhost:9083/connectors/pedidos-
realizados-connector
rodrigovieirapinto@MacBook-Pro-de-Rodrigo docker %
./pedidos-realizados.sh
{"name":"pedidos-realizados-connector","config":
{"connector.class":"io.debezium.connector.mongodb.Mongo
DbConnector","mongodb.connection.string":"mongodb://use
r:password@mongo:27017/?
replicaSet=rs0","mongodb.name":"restaurantes","database
.include.list":"restaurantes","collection.include.list"
:"restaurantes.pedidos","topic.prefix":"mongo","key.con
verter":"org.apache.kafka.connect.storage.StringConvert
er","value.converter":"io.confluent.connect.avro.AvroCo
nverter","value.converter.schema.registry.url":"http://
schema-
registry:8081","value.converter.auto.register.schemas":
"false","value.converter.schemas.enabled":"false","tran
sforms":"unwrap","transforms.unwrap.type":"io.debezium.
connector.mongodb.transforms.ExtractNewDocumentState","
transforms.unwrap.drop.tombstones":"true","name":"pedid
os-realizados-connector"},"tasks":[],"type":"source"}
```

Com essas alterações, podemos agora enviar pedidos com ou sem o complemento no endereço. A mensagem chegará normalmente no Kafka, graças ao Kafka Connect e ao aval do Schema Registry. Caso uma mensagem com *schema* diferente seja enviada, ela não será aceita pelo Kafka.

Por exemplo: imagine que, por uma razão qualquer, as mensagens passem a ter também um campo de

`pais` no `endereco`. Se colocarmos esse campo na classe `Endereco` no *ms-pedidos*, e realizarmos mais um pedido, a persistência no MongoDB será feita sem problemas.

Confira essa alteração a seguir:

```
@EqualsAndHashCode
public class Endereco {
    private String rua;
    private Integer numero;
    private String bairro;
    private String cidade;
    private String estado;
    private String cep;
    private String complemento;
    private String pais;

    public Endereco(String rua, int numero, String
bairro, String cidade, String estado, String cep,
String complemento) {
        this.rua = rua;
        this.numero = numero;
        this.bairro = bairro;
        this.cidade = cidade;
        this.estado = estado;
        this.cep = cep;
        this.complemento = complemento;
        this.pais = "Brasil";
    }

// trechos de código omitidos por brevidade
```

Entretanto, assim que a mensagem chega ao Kafka Connect, ocorre o seguinte erro (trechos omitidos por brevidade):

```
2025-05-31 17:40:00.215 | [2025-05-31 20:40:00,215]
INFO 172.18.0.5 - - [31/May/2025:20:40:00 +0000] "GET
/connectors?expand=info&expand=status HTTP/1.1" 200
1142 "-" "Redpanda Console" 8
(org.apache.kafka.connect.runtime.rest.RestServer)
2025-05-31 17:40:05.740 | [2025-05-31 20:40:05,739]
INFO 1 record sent during previous 00:12:05.497, last
recorded offset of {server_id=mongo} partition is
{sec=1748724005, ord=1,
resume_token=zQAAAAJfZGF0YQC9AAAAODI2ODNCNjkyNTAwMDAwMD
AxMkIwNDJDMDEwMDI5NkU1QTEwMDQ2N0ZEMTI1RTQ1MDY0MzA2OUQ0Q
TdERENCMUZBOEQ0NjQ2M0M2RjcwNjU3MjYxNzQ2OTZGNkU1NDc5NzA2
NTAwM0M2OTZFNzM2NTcyNzQwMDQ2NjQ2RjYzNzU2RDY1NkU3NDRCNjU
3OTAwNDY2NDVGNjk2NDAwNjQ2ODNCNjkyNEMzRjhCRjdBN0UzREY4Mj
MwMDAwMDQAAA==}
(io.debezium.connector.common.BaseSourceTask)
2025-05-31 17:40:07.050 | [2025-05-31 20:40:06,994]
ERROR WorkerSourceTask{id=pedidos-realizados-connector-
0} Task threw an uncaught and unrecoverable exception.
Task is being killed and will not recover until
manually restarted
(org.apache.kafka.connect.runtime.WorkerTask)
2025-05-31 17:40:07.050 |
org.apache.kafka.connect.errors.ConnectException:
Tolerance exceeded in error handler

...

2025-05-31 17:40:07.051 | Caused by:
org.apache.kafka.connect.errors.DataException: Failed
to serialize Avro data from topic
```

```
mongo.restaurantes.pedidos :
2025-05-31 17:40:07.051 |     at
io.confluent.connect.avro.AvroConverter.fromConnectData
(AvroConverter.java:107)
2025-05-31 17:40:07.051 |     at
org.apache.kafka.connect.runtime.AbstractWorkerSourceTa
sk.lambda$convertTransformedRecord$6(AbstractWorkerSour
ceTask.java:491)
2025-05-31 17:40:07.051 |     at
org.apache.kafka.connect.runtime.errors.RetryWithTolera
nceOperator.execAndRetry(RetryWithToleranceOperator.jav
a:208)
2025-05-31 17:40:07.051 |     at
org.apache.kafka.connect.runtime.errors.RetryWithTolera
nceOperator.execAndHandleError(RetryWithToleranceOperat
or.java:245)
2025-05-31 17:40:07.051 |     ... 13 more
2025-05-31 17:40:07.051 | Caused by:
org.apache.kafka.common.errors.SerializationException:
Error retrieving Avro
schema{"type":"record","name":"pedidos","namespace":"mo
ngo.restaurantes","fields":[{"name":"_id","type":
["null","string"],"default":null},
{"name":"itensDoPedido","type":["null",
{"type":"array","items":["null",
{"type":"record","name":"itensDoPedido","namespace":"mo
ngo.restaurantes.pedidos","fields":
[{"name":"item","type":["null",
{"type":"record","name":"item","namespace":"mongo.resta
urantes.pedidos.itensDoPedido","fields":
[{"name":"numero","type":
["null","long"],"default":null},{"name":"nome","type":
["null","string"],"default":null},
{"name":"descricao","type":
["null","string"],"default":null},
{"name":"preco","type":
```

```
["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido.item"}],"default":null},{"name":"quantidade","type":["null","int"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.itensDoPedido"}]}],"default":null},{"name":"endereco","type":["null",{"type":"record","name":"endereco","namespace":"mongo.restaurantes.pedidos","fields":[{"name":"rua","type":["null","string"],"default":null},{"name":"numero","type":["null","int"],"default":null},{"name":"bairro","type":["null","string"],"default":null},{"name":"cidade","type":["null","string"],"default":null},{"name":"estado","type":["null","string"],"default":null},{"name":"cep","type":["null","string"],"default":null},{"name":"pais","type":["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos.endereco"}],"default":null},{"name":"total","type":["null","string"],"default":null},{"name":"_class","type":["null","string"],"default":null}],"connect.name":"mongo.restaurantes.pedidos"}

...

2025-05-31 17:40:07.051 | Caused by:
io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException: Schema not found; error code: 40403
```


É possível notar no erro descrito que o campo `pais` consta no *schema* da mensagem. Entretanto, como o

Schema Registry não possui *schemas* cadastrados para esse tópico que possuam essa informação, a mensagem é rejeitada.

9.5 Resumo

Neste capítulo, vimos como o Schema Registry pode ser integrado ao Kafka Connect. Entendemos que, mesmo diante de questões relacionadas à compatibilidade, o Schema Registry cumpre o seu papel ao impedir que mensagens fora do formato adequado sejam inseridas nos tópicos Kafka.

Concluímos, assim, a construção do Alura Eats que nos acompanhou ao longo do livro. Ao explorar diferentes formatos de serialização (como Avro, Protobuf e JSON Schema) e integrá-los ao ecossistema Kafka, pudemos observar como o Schema Registry ajuda a garantir consistência e interoperabilidade entre produtores e consumidores.

No próximo capítulo, vamos refletir sobre os aprendizados e caminhos possíveis para evoluir com o uso dessa ferramenta em ambientes reais.

Capítulo 10
Conclusão

Ao longo deste livro, partimos de um cenário no qual as mensagens eram trocadas entre aplicações sem controle formal de contratos e avançamos até uma arquitetura robusta e confiável com o uso do Schema Registry.

Começamos com mensagens JSON, nas quais a ausência de um esquema validado em tempo de envio ou consumo pode gerar problemas silenciosos, difíceis de diagnosticar. Em seguida, evoluímos para o uso do Avro, explorando seu suporte nativo a versionamento e serialização compacta. Por fim, experimentamos o Protobuf, destacando sua integração com ambientes mais diversos.

Durante essa jornada, criamos produtores e consumidores que se beneficiaram do Schema Registry como fonte única de verdade para os contratos de dados. Exploramos a construção de desserializadores próprios, aprendemos como garantir conformidade entre versões e descobrimos recursos valiosos como o controle de compatibilidade e o gerenciamento de múltiplos tipos de *schema*.

Também dedicamos um capítulo à API do Schema Registry, que nos permitiu interagir diretamente com a ferramenta, automatizar processos, registrar e buscar *schemas* programaticamente. Este recurso é essencial para integrações avançadas e *pipelines* de dados mais dinâmicos.

Encerramos com a integração do Schema Registry ao Kafka Connect, ampliando sua utilidade para integrações com sistemas externos de forma estruturada e governada.

Mais do que um recurso técnico, o Schema Registry se mostrou essencial para que pudéssemos confiar nos dados transmitidos, pois promove contratos claros, detecta inconsistências precocemente e permite a evolução segura dos sistemas ao longo do tempo.

Este livro aborda uma gama de temas essenciais e práticos, fornecendo uma base consistente para o trabalho com Schema Registry e sua integração aos sistemas de dados. Entretanto, existem áreas avançadas que poderão ser aprofundadas à medida que a jornada do leitor se desenvolve, como controle de acesso a *schemas*, versionamento automatizado via CI/CD e governança dos dados.

Espero que esta leitura tenha oferecido não apenas conhecimento técnico, mas também uma nova perspectiva sobre o papel dos contratos de dados em sistemas distribuídos. Que este aprendizado seja útil em seus projetos atuais e sirva de base para futuras soluções mais confiáveis e escaláveis.

Capítulo 11

Bibliografia

EVANS, Eric. *Domain Driven Design*: Atacando as Complexidades no Coracao do Software. 3. ed. Brasil: Alta Books, 2016.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. United States: University of California, 2000.

GAMMA, E. et al. *Padrões de Projetos*: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman Companhia Ed, 2006.

HOHPE, Gregor; WOOLF, Bobby. *Enterprise Integration Patterns*: Designing, Building, and Deploying Messaging Solutions. 1. ed. Boston Munich: Addison-Wesley Professional, 2003.

PINTO, R. V. *Kafka Connect e Kafka Streams*: Integre suas aplicações ao Kafka usando as melhores práticas. 1. ed. Brasil: Casa do Código, 2024.

SANTANA, E. F. Z. *Apache Kafka e Spring Boot*: Comunicação Assíncrona Entre Microsserviços. 1. ed.

Brasil: Casa do Código, 2022.

VERNON, Vaughn. *Implementando Domain-Driven Design*. 1. ed. Brasil: Alta Books, 2016.